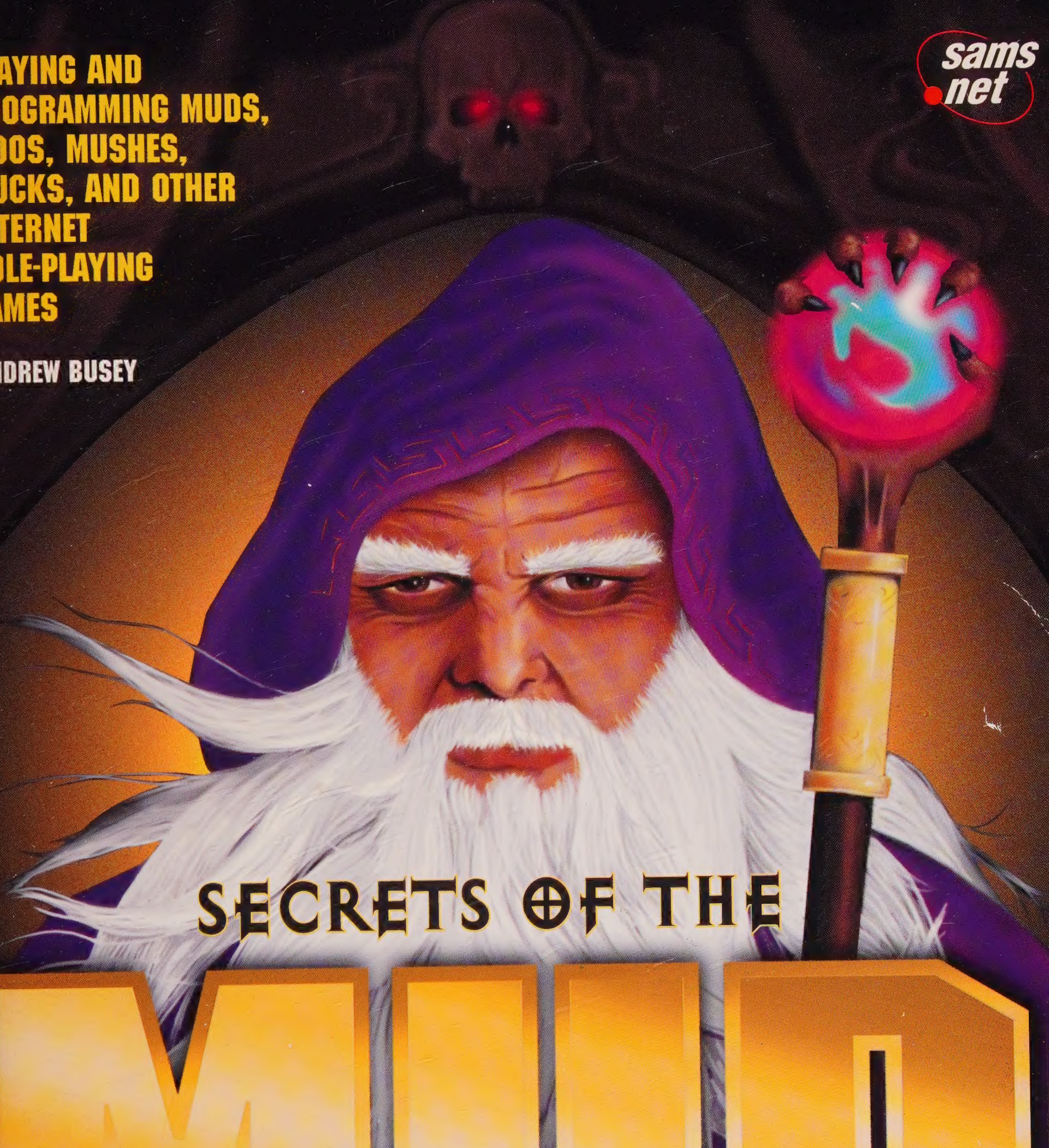
sams.net

PLAYING AND PROGRAMMING MUDS, MOOS, MUSHES, MUCKS, AND OTHER INTERNET ROLE-PLAYING GAMES

ANDREW BUSEY



SECRETS OF THE MUD WIZARDS

Basic MUD Commands

Command	*Abbreviation*	*Description*
`say <message>`	`"<message>`	Says `<message>` to everyone in your current environment.
	`-<name> <message>`	Says `<message>` to all in your current environment, but directed to `<name>`.
`emote <message>`	`:<message>`	Emotes the given `<message>` in the form `Your_name <message>`.
	`::<message>`	A special emote in the form of `Your_name<message>`. (No space.)
`page <player> <message>`	`'<message>`	Sends a private `<message>` to `<player>` anywhere on the MOO.
`whisper <message> to <player>`		Whispers a private `<message>` to `<player>` in the same room.

Player Set-Up

`@describe me as <description>`		Other players will see `<description>` when they `look` at you.
`@gender <gender>`		Sets your character's gender. `<gender>` is most likely `male` or `female`.
`@password <old password> <new password>`		Changes your character's password.
`look <object>`		Looks at `<object>` to see its description.
`inventory`	`i`	Sees the objects you currently have in your possession.
`@who <player>`		Sees who is currently logged on or if a specific `<player>` is on.

SECRETS OF THE MUD WIZARDS

Andrew Busey

sams.net

201 West 103rd Street
Indianapolis, Indiana 46290

International Standard Book Number: 0-672-30723-5

Library of Congress Catalog Card Number: 95-67664

98 97 96 95 4 3 2 1

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 95-1 shows that the first printing of the book occurred in 1995.

Composed in Stone and MCPdigital by Macmillan Computer Publishing

Printed in the United States of America

President, Sams Publishing	*Richard K. Swadley*
Publisher, Sams.net	*George Bond*
Acquisitions Manager	*Greg Wiegand*
Development Manager	*Dean Miller*
Managing Editor	*Cindy Morrow*
Marketing Manager	*John Pierce*

Acquisitions Editor
Mark Taber

Development Editor
Angelique Brittingham

Production Editor
Jill D. Bond

Copy Editor
Anne Owen

Editorial/Graphics Coordinator
Bill Whitmer

Formatter
Frank Sinclair

Technical Reviewer
Jennifer Smith

Cover Designer
Tim Amrhein

Cover Illustrator
Jeff Yesh

Book Designer
Alyssa Yesh

Page Layout
Charlotte Clapp, Mary Ann Cosby, Terrie Deemer, Louisa Klucznik, Ayanna Lacey, Casey Price, Tina Trettin, Susan Van Ness, Dennis Wesner

Proofreading
Georgiana Briggs, Mona Brown, Mike Henry, Kevin Laseau, Paula Lowell, Brian-Kent Proffitt, SA Springer

Indexer
Cheryl Dietsch

Overview

Contents

Acknowledgments

First off, I'd like to thank my parents for their support. I know, that's the typical "I'm a new author and have to thank my parents kind of thing," but I think they really do deserve it. They were responsible for getting me my first computer and, probably much to their dismay, my first modem. And without all the fun (and enormous phone bills), I probably wouldn't have learned as much as I have about computers and gotten so involved in the online world. So thanks Mom and Dad. Also, thanks to my sister Alyson who, along with my parents, got to hear all of my gripes while I was writing this book.

Thanks to Mark Taber at Sams and Angela Gunn (who wrote *Plug-n-Play Mosaic*) for helping to convince me to write this book. Thanks again to Mark Taber and to Jill Bond for their patient assistance in helping me learn the ropes on my first complete book. And thanks to all the other people at Sams who have helped make this a better book.

Special thanks goes to Frank "Bleys" Stevenson, Jr., who got me hooked on MUDs—you'll notice the "Bleys" nickname; that is his MUDname, which he actually uses in real life now. Frank contributed a lot to this book and got a lot of work done in a hurry. Thanks also to my other guest authors, Peter Novosel, Joseph Poirier, and Chris Stacy, all of whom (along with Bleys) did a great job and helped make this book a good one.

Jennifer Smith also deserves special recognition; not only for her contributions to this book, but also for her service to the MUD community. She maintains the MUD FAQ (Frequently Asked Questions) list, which is a great help to new MUDders. Jennifer, thank you for allowing me to quote from the FAQ throughout this book to provide a different perspective. Further, thank you for helping to keep me from focusing too much on LPMUDs and DikuMUDs, and for doing a great job of tech editing this book!

I want to thank the manufacturers of the computer games *Warcraft* and *Warlords* for providing much needed interruptions from working on this book. I also want to thank the ImagiNation Network, AgChat in Austin (telnet `chat.eden.com 2317`) and all the MUDs out there, for giving me somewhere to hang out and play or chat when I needed a break. And, finally, the theme music for the book consists of: Depeche Mode, REM, Sarah McLachlan, The Cranberries, Offspring, the Moon Seven Times, *In the Name of the Father, Pulp Fiction,* and *Interview with the Vampire* sound tracks.

About the Authors

Andrew Busey is Vice President of Business Development and a co-founder of Net-Link, Ltd., which develops and markets easy-to-use integrated Internet software and services. In the past, he also was project manager and one of the authors for the *World Wide Web Yellow Pages* from New Riders Publishing. In his recent past, he has among other things, been the product manager for Mosaic at Spyglass and co-founded New Media Publications, Inc., which publishes *Melvin* (`http://www.melvin.com/`)—one of the most fun sites on the World Wide Web. Andrew has been using the Internet for about eight years, MUDding for about four, and been a god on at least one MUD for about three of those years. He received his degree from Duke University where he studied computer science and marketing. He currently lives in Austin, Texas, with his Siamese cat, Ashen. You can contact Andrew via e-mail at `busey@eden.com`.

Joseph Poirier (`snag@holli.com`) is a software engineer for Network Design Technologies, Inc., where he designs and implements object-oriented telecommunications network optimization software. He graduated from Purdue University in 1990 with a B.S. in Computer Science. Known as Snag on several MUDs, he can frequently be found playing cards in the virtual poker halls. He is the one wearing the bunny slippers. (See Chapter 15, "MUSH and MUCK Programming.")

Introduction

In the last several years the Internet has grown faster than anyone expected. This growth largely has been fueled by people's desire to communicate with each other and a desire to quickly retrieve information. E-mail, Net news, FTP, and the World Wide Web have all provided powerful ways for users to interact with each other and with information.

But, none of these are *real time*. When one uses these traditional Internet tools to communicate with others, the communication is delayed. If I send you e-mail, for example, the e-mail will meander through the Internet, eventually arriving at your computer. But you may not read it for an hour or even a week, and you may not respond to it for another hour. MUDs are a lesser-known Internet resource that does allow users to interact in real time. Real time means that as soon as I type something (like a message to you), you will see it. And you can respond instantly. Another example of real time is talking on the telephone.

MUD stands for Multi-User Dungeon, but do not let the name scare you. MUDs are far more than dungeons. Many MUDs have absolutely no relation to dungeons. In fact, some people are starting to call it by other names such as *social virtual reality*. This of course implies that the system has to do with virtual reality. Many still conjure up images of people wearing funny helmets and waving their arms at nothing as the image of someone in virtual reality. Well MUDs are virtual reality, but they are from the other side of the spectrum. The focus of MUDs is to create *virtual worlds*.

A virtual world, in the MUD sense, is a creation that users can walk around in and interact with other inhabitants and objects in the virtual world. Those inhabitants may be other users or they may be computer generated. Each MUD has its own virtual world with its own unique geography and inhabitants. Much of the fun of a MUD is exploring this virtual world.

The other word used was *social*. MUDs are inherently social—very much the theme parks and pick-up bars of the Internet. I say theme parks because they provide virtual worlds for the user to explore and enable the user to assume a new identity—to escape reality into this virtual world. MUDs resemble bars in the sense that people often meet on MUDs, both in platonic and romantic ways. MUDs truly are social, providing one of the best mediums for people to meet each other and have fun together on the Internet.

MUDs also provide a framework for games and role-playing. Much of this is set in worlds that already have been created like those from *Star Trek*, *Star Wars*, Anne McCaffery's *Dragonriders of Pern*, *The Vampire: The Masquerade*, *Shadowrun*, and many more. You can explore virtual worlds that mimic these popular books and movies, and explore others that are original or based very loosely on previous ideas.

While you role-play or game in one of these worlds, you probably will team up with other players to be more successful in the things you do. It's easier on game-oriented MUDs to advance more quickly if you ally yourself with other players. And if nothing else, you certainly will want to talk to more experienced players to learn the secrets they may have accumulated and get a better idea of the best places to go.

Socializing is a big part of MUD culture, thus, social aspects and adventuring in a new virtual world are two areas of interest. While there are many other reasons people MUD, there is one more of particular interest. On a MUD, a player eventually can become a wizard or even a god. This gives players the ability to expand the virtual world in their own way. Once a player has achieved this level, he or she can leave a personal signature on the MUD by adding to its world. This sense of power and the capability to create also are attractive to many players.

Who Should Read This Book?

This book is designed to be useful for anyone who wants to MUD and those who already are MUDding. Perhaps you just read an article about MUDs in *Newsweek* that was very interesting and want to learn more, or maybe you have been MUDding for a while now. This book is divided into three sections that anyone who has an interesting in MUDding should find interesting.

Part I: An Introduction to MUDs

Part I covers what MUDs are, how they work, what they look like, and how to connect. This section should be helpful to anyone who is vaguely familiar with the Internet but has yet to try a MUD.

Part II: MUD Player's Guide

Part II goes into greater detail about the different types of MUDs, focusing on LPMUDs, DikuMUDs, MOOs, MUSHes, and MUCKs. For the experienced MUDder, it should be useful as a command reference and a way to learn about the other types of MUDs. For the novice, it provides a step-by-step guide through the different types of MUDs and their commands.

Part III: MUD Programming Guide

Part III discusses programming on different types of MUDs. The novice will not have an immediate need for much of this section, but will find it helpful the more he or she MUDs and eventually becomes a wizard. The experienced MUDder who does not yet know how to program will find this section an invaluable tool for learning the programming methods for the different MUDs. Even experienced MUD programmers may learn something new!

The Appendixes

The appendixes contain a MUD directory, a glossary of MUD terms, a list of MUD clients, and a list of available servers.

The Terminology of the Book

This book uses several different fonts and boxes to differentiate ideas that are introduced. This segment will explain the various things you will want to watch out for.

Text and Fonts

Certain special fonts and text are used to differentiate certain commands and screen images.

COMMAND

This format is used to introduce new MUD commands. It includes the MUD command, an explanation, and syntax for that particular command.

NOTE

Notes are used to introduce ideas of special interest and to explain ideas in more detail. Notes also are used to set apart the explanations of things that happen in large "virtual tours," which use many examples that come straight from a MUD.

TIP

Tips separate and highlight information that is designed to help players perform better. They are focused primarily on combat MUDs.

WARNING

Warnings are used to call your attention to actions that can be dangerous. Warnings also are used to caution you about certain material that may be offensive.

```
Shadowed monospace code indicates sample sessions taken directly from a MUD.
```

Icons

Throughout the book, several icons are used to designate areas of interest to a specific segment of the readership. For example, you will find there are many different types of MUDs and that you will focus on only one or two of the different types. Icons are used to set up the information that pertains to the topic designated by the icon.

LP MUD This icon designates that what follows is relevant specifically to LPMUDs.

DIKU MUD This icon designates that what follows is relevant specifically to DikuMUDs.

MOO This icon designates that what follows is relevant specifically to MOOs.

MUSH This icon designates that what follows is relevant specifically to MUSHes.

MUCK This icon designates that what follows is relevant specifically to MUCKs.

Conventions Used in This Book

The following typographic conventions are used in this book:

- Code lines, commands, statements, variables, and any text you type or see on the screen appears in a `computer` typeface.
- Placeholders in syntax descriptions appear in an *`italic computer`* typeface. Replace the placeholder with the actual filename, parameter, or the element it represents.
- User input appears in **`bold monospace`**. This represents text you type.
- *Italics* highlight technical terms when they first appear in the text and sometimes are used to emphasize important points.

I PART

Introduction To MUDs and MUDding

I

CHAPTER

Introduction to MUDs

MUD is an acronym for either *Multi-User Dungeon* or *Multi-User Dimension.* Although there are many different kinds of MUDs, this book concentrates on the five most popular kinds of MUDs. They are as follows:

LPMUD—[**L**ars **P**ensj| **M**ulti-**U**ser **D**ungeon

DikuMUD—**D**atalogisk **I**nstitut **K**oebenhavns **U**niversitet (Department of Computer Science, University of Copenhagen) **M**ulti-**U**ser **D**ungeon

MOO—**M**UD **O**bject **O**riented

MUCK—This name is taken from MUD, but it has no particular meaning.

MUSH—**M**ulti-**U**ser **S**hared **H**allucination

Although this book focuses on these five different types of MUDs, you will find that much of what the book teaches you can be applied to any MUD-like system. This book also includes lists of some sample MUDs on the Internet and their descriptions. You get tips for playing and programming MUDs and receive some insight into the uses of MUDs. You also learn about other systems on and off the Internet that are related to MUDs, such as *IRC* (the *Internet Relay Chat*) and commercial multi-user games that resemble MUDs.

MUDs have become a steadfast part of the Internet as an outlet for fun and socializing. Rivaled only by IRC, MUDs provide the leading forum for real-time interaction on the Internet. In "real-time" interaction, the user can interact with other users immediately. This interaction is different from e-mail and netnews (Usenet), which offer users the opportunity to interact with each other but with time delays. E-mail and news are the two most common uses of the Internet because they are accessible via the major online services. With e-mail and news, the user must wait for a reply to messages. This means that correspondence can have delays of days or even weeks, depending on when someone chooses to respond, although once a response is sent, it usually takes only minutes or at most a few hours to reach its destination.

In today's world, everybody wants instant gratification. On MUDs, users talk to each other, and their correspondence is relayed in milliseconds—nearly as fast as your voice is relayed during a phone call. Correspondence is limited only by the typing speeds of those corresponding. The only other Internet services that are easily accessible and that provide real-time interaction are the `talk` command and IRC. The `talk` command does not offer the opportunity to meet new people. It functions only as a mechanism for two people that already know each other to communicate—much like a phone call. IRC, on the other hand, is a veritable medley of people talking about every imaginable subject. Averaging over a thousand users at any given time, IRC covers a broad spectrum of topics. However, IRC is organized loosely, having different channels that focus on predetermined topics.

MUDs, on the other hand, create a virtual world where the users or players can interact with each other, wander on their own, and talk one-on-one. When logging in to a MUD for the first time, you choose a name (usually not your real name) for your MUD character. When on a MUD, you have a virtual body. You can give this body any name and any description; you create your "persona." Your new MUD character doesn't have to be you. In fact, you have the power to create and live a new life. You can assume any personality you like, but more on this in Chapter 3.

The Internet contains hundreds of MUDs, and they are nearly all distinct in one way or another. Therefore, this book has a huge amount of material to cover. This book covers topics of interest for the novice and the expert.

Different Types of MUDs

Some of the terms used here will become more familiar as you read. Many are covered in much more detail in Chapter 3, but because these definitions are helpful throughout the book they are introduced here.

- *Combat MUDs* are MUDs that have built-in systems that support combat between players and monsters. Combat MUDs make up a large portion of the MUDs. Monsters are creatures the computer controls and that fight with characters the players control. LPMUDs, DikuMUDs, and some MUSHes, fall into this category.
- Social MUDs are those MUDs that do not have a built-in gaming system like those found on Combat MUDs. MOOs, MUCKs, and most MUSHes, also fall into this category.

LP MUD

LPMUD is a type of MUD that uses a built-in combat system and has a particular look and feel to it that you will learn to identify in the next two chapters. There are more LPMUDs than any other type of MUD. Many LPMUDs have a fantasy orientation, but because they are very modifiable, they have been adapted to many other genres. Only wizards can program or create objects on an LPMUD. This type of MUD is covered in much more detail in Chapter 8 and programming LPMUDs is covered in Chapter 13.

DIKU MUD

DikuMUDs are another form of MUDs that have a built-in combat system. They are similar to LPMUDs, but with less flexibility for development but (arguably) better combat systems. DikuMUDs tend to be fantasy oriented. Only wizards can program or create objects on a DikuMUD. For more information about DikuMUDs, see Chapter 9.

MOO

MOOs are primarily social MUDs in which players tend to hang out and chat. In general, every MOO user is allowed to create objects within the MOO, so many people go to MOOs to create. LambdaMOO is probably the single most popular MUD on the Internet. MOOs are covered in detail in Chapter 7 and programming MOOs is covered in Chapter 15.

MUSH

MUCK

MUSHes and *MUCKs* are very similar and share many qualities with MOOs. All three were derived from the same originial source—TinyMUD. Some MUSHes have been modified and turned into combat MUDs, although they have a different feel than that found on the more traditional combat MUDs (LPMUD and DikuMUD). On many MUSHes and MUCKs, players are allowed to create objects, although sometimes special permission is required. Generally, MUSHes and MUCKs are social MUDs; however, they often have a tendancy toward more in-depth role-playing. For more information about programming MUSHes and MUCKs, see Chapter 15.

WARNING

Before jumping into the thick of things, I want to give a word of caution. MUDs are *extremely* addictive. There are very few things that are as psychologically addictive as MUDs. This addictiveness is discussed at greater length in the book. Before you start MUDding, you must realize that the potential for addiction is there. For more information, see the section called "Addictiveness" later in this chapter.

What Is a MUD?

A MUD is a form of a virtual world. While MUDs do not yet have the glamorous graphics of virtual reality, they do have their own allure that is just as unique. MUDs weave a virtual world around the user by providing a first-person perspective of one's environment. MUDs lack the graphical appeal of other user interfaces, but most people can easily grasp the MUD user interface because it's like reading a book.

Following is an example of a MUD conversation:

```
John says, "Hey, what's up?"
Helen says, "Not much. What about you?"
Helen smiles at John.
John says, "I'm fine."
John smiles at you.
John says, "You must be new. Hi, I'm John."
John shakes hands with you.
Helen says, "Don't worry, you'll get used to it."
Helen pats you on the back.
```

Notice that it's as if the whole conversation is from your perspective. Following is how John sees the conversation:

```
You say, "Hey, what's up?"
Helen says, "Not much. What about you?"
Helen smiles at you.
John says, "I'm fine."
You smile at Newbie.
You say, "You must be new. Hi, I'm John."
You shake hands with Newbie.
Helen says, "Don't worry, you'll get used to it."
Helen pats Newbie on the back.
```

In the preceding conversation, you are given the name `Newbie`. On MUDs, a *newbie* is a new user. It's similar to using "freshman" or "frosh" as the name for a new college student. In the conversation, John is also experiencing the conversation from a first-person point of view. Each person on the MUD sees the world from his or her own perspective.

You can use MUDs in many ways, but this book focuses on using them for entertainment. However, MUDs are beginning to be used in schools as a learning and social tool, and growth in this area is likely to continue.

Combat MUDs, which usually are LPMUDs and DikuMUDS, have a framework for gaming built into them; MOOs, MUCKs, and MUSHes generally don't. A typical combat MUD world is inhabited by monsters of all kinds, and the players travel around the world slaying these monsters for treasure and experience. By accumulating experience, players can advance to different levels and learn new skills, allowing them to kill even larger monsters. Players can often choose to be a *mage* (a type of character that can use magic), *priest*, *fighter*, *thief*, or any number of other roles. The role that players choose varies widely depending on the individual MUD. This topic is discussed in much greater detail in Chapter 3.

When you log in to a MUD and wander around, you are likely to encounter other users who are also wandering about the virtual world. You may pass people on the street or meet them in a store. Meeting other people leads to quite a bit of social interaction, such as users teaming up to kill monsters together, people getting "MUD married" (more on this in Chapter 6), and just general conversation. Because no one can see you on a MUD, people are more willing to talk to total strangers. It's pretty safe to talk to someone through miles of wire. Unlike in real life, on MUDs it's more acceptable to just walk up to people and start talking to them. All of these factors lead to a much greater level of interaction and overall fun on MUDs. Lots of interesting conversations can spring up.

Two components on many MUDs are *players* and *wizards*. Players make up the majority of users on most MUDs. A player is the basic MUD user. He or she can move through the MUD, interact with other users, and, on combat MUDs, kill monsters. The second class of user is the wizard (also known as *administrator*, *builder*, *god*, or *elder*). Wizards have the ability to create monsters, rooms, and objects. Wizards can alter players and can affect the MUD world in any way they want to. On some MUD derivatives, like MOOs, all users can build new objects; but on most MUDs, this capability is restricted to wizards.

Why MUD?

People "MUD" for many reasons. The allure of creating one's own world, or even just living in a world with different rules, can be immense. Politics, adventure, and the brave new world are great attractions. People like the fact that they can spin a new reality or shed the boundaries of an everyday world.

In the MUD world, no one knows whether you are rich or poor, black or white, male or female, and all the encumbrances of the real world are gone. However, if you change the reality that is you too much, you may find yourself in a web you cannot spin your way out of. Many find this fine line between reality and the virtual world to be the most important draw for them.

Because all MUDs have politics, you can be pulled into the political aspects of MUDs. The politics of LPMUDs often relate to how powerful players are within the game atmosphere and their ability to organize others into groups that may feud or work together. Most MOOs have something called the *Architecture Review Board*, which oversees new real estate and much more. In fact, LambdaMOO (the most well-known and populated MOO) has developed its own governmental system. For example, this system proposes referendums on important issues, and the populace may vote on them.

On combat MUDs (MUDs that have a built-in game system, similar to role-playing games like Dungeons and Dragons), there are many more reasons to play. On these systems, you can build up a character and expand his or her abilities as you play, thereby becoming more and more powerful within the game. Some people enjoy this power for the sake of advancing; others want the power to kill creatures in the MUD on a whim. Still others desire the influence that being powerful gives them over other players. Some people just like to hang out and talk.

Hanging out? That seems an odd thing to do online, but MUDs are good places to meet people. While it may be hard to walk up to a stranger in a bar, it's very easy on a MUD. All conversations and meetings between MUDders is easy, both male and female, platonic and romantic. Everyone expects random conversations and weird things on MUDs, and no one has self-confidence problems. It is easy to talk online—especially on MUDs.

MUDs are like theme parks; they come close to being the theme parks of the Internet. Disney World and Six Flags take us away from reality into a variety of re-created worlds with their own themes. MUDs provide the same function on the Internet by breaking the monotony and the everyday humdrum of e-mail, FTP (the file transfer protocol used for transfering files on the Internet), and even the World Wide Web. MUDs break the monotony with interaction, the cability to chat with others, and the ability to escape into an imaginary world.

Whether that world is taken from a book like Anne McCaffery's *Dragonriders of Pern* or Robert Jordan's *Wheel of Time*, MUDs provide a wide variety of environments that can't be duplicated or visited any other way. These wonderful worlds, combined with a vast array of accessible people, come together to make MUDs a very great place to visit (and for some, a good place to live)...which leads us to the issue of how addictive MUDs can be.

Addictiveness

As I mentioned at the beginning of the chapter, MUDs can be very addictive. This section discusses some of the reasons that MUDs can be so addictive. Knowing why MUDs are addictive may prevent you from becoming addicted yourself. (I know, it can't happen to you. We've all said that before.) Following are the reasons that I think MUDs are addictive:

- **Social Forces**: This reason is pretty simple. You meet a bunch of people online, and you end up talking to them for a long time. It seems boggling, but it is really that simple. This effect can also be observed in the college environment. In college, groups of people stay up talking about things until ridiculously late hours, often because the conversation is interesting and no one wants to leave. People may not want to leave because they want to give their opinions, the conversation is just too interesting, or perhaps they think it may be rude to leave. It is impossible to understand this concept until the first time you see the morning sun come through your window after MUDding all night.

- **Political/Competitive Reasons**: Often (primarily on gaming MUDs) there are several new players on a MUD at any given time, and these players tend to bond into a group. This group then develops into a sort of political organization (informal) with its own motives. Perhaps a couple of players in the same class or guild start working together on a regular basis toward some goal. This group goal drives everyone involved to be on the MUD constantly so they can possibly be the leader of the group. Members of the group also stay on the MUD to make sure they remain equal in level and status to everyone else in the group.
- **Temporary Drive**: This temporary drive happens at different times and keeps you online for more time than you expected. An example of this is that when you die on most MUDs, you lose a lot of your experience points and generally get mad. Then you decide to stay on the MUD until you regain all the experience you've lost. More often, you manage to get an incredible weapon or piece of armor you can't bear to lose, so you keep playing until you pass out. The same thing can happen if someone significantly more powerful than you starts helping you. It becomes much easier to advance, so you want to stay on as long as the person helping you continues to help you.

Is MUDding a Game or an Extension of Real Life with Gamelike Qualities?

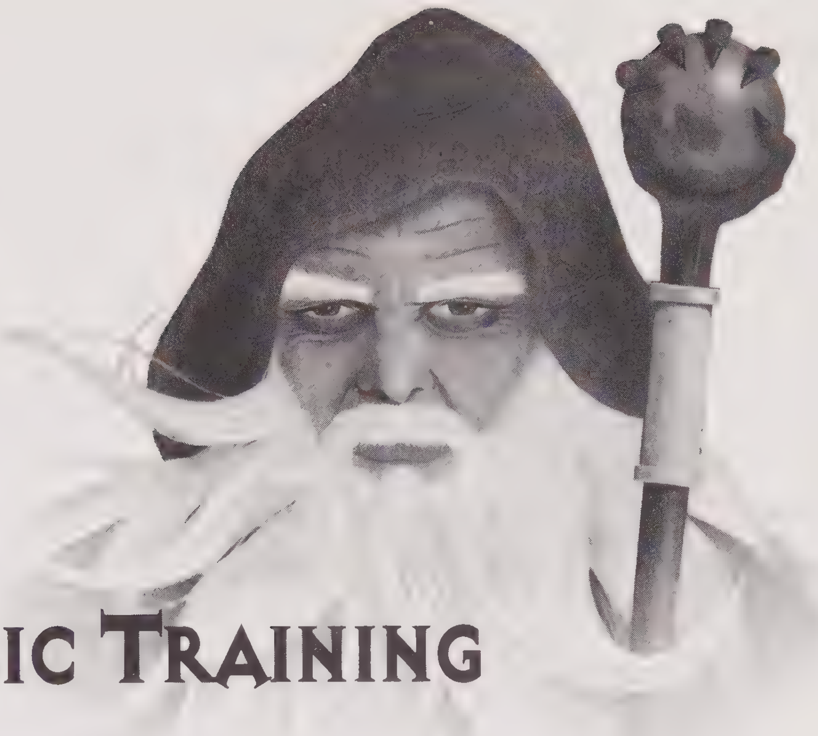
It's up to you. Some jaded cynics like to laugh at idealists who think that MUDs are partially for real, but we idealists think they're not playing right. Certainly the hack-n-slash stuff is only a game, but the social aspects may well be less so.

Summary

Now you have a little bit of an idea of what a MUD is and how it works...and even why you may want to play one. You also know how the book is formatted. I wish you luck with your adventures into these new worlds called MUDs. The rest of the book provides you with the tools you need to get started on MUDs quickly and to be successful in whatever you hope to achieve on them.

2

CHAPTER



MUD Basic Training

The purpose of this chapter is to run through the basics of MUDding. This entails showing you how MUDs work, the basic interface that MUDs use, and an introduction to basic MUD commands. This chapter focuses primarily on teaching you how to navigate MUDs and gives you an idea of how MUDs work. The commands for interacting with other players are covered in Chapter 5.

So prepare to enter MUD boot camp, where you will get your feet wet as you learn the secrets of navigating MUDs!

Using a MUD

If you have ever played a text-based adventure game such as the classic Infocom games (Zork, for example), you immediately will be familiar with the basic interface MUDs use. Because MUDs have no graphical aspect, they use words to convey the virtual worlds they portray.

The smallest unit of MUD geography is the room. Imagine a piece of graph paper and then imagine each box as a room. This is the way a MUD works. You move from box to box, encountering when you enter it whatever else might inhabit that square. Because of this, it is easy to map a MUD.

Prompts: When Do I Type?

Because you probably will interact with MUDs on a regular basis, it is important to know when you can enter information and what the MUD might be doing. Because different types of MUDs deal with user input in different ways, this section briefly covers user input before showing you how the MUD world works and how to navigate through it.

In MS-DOS you often see the `C:\>` prompt (or something similar), and in UNIX you probably see the `%`. When you see these prompts, you know that you then can enter a command. If you use the `DIR` command in MS-DOS, for example, you will see the directory of files. When it has been completely displayed, you return to the prompt and can enter another command. Because MUDs are interactive, they do not work in quite the same way; you could receive information from another source that acts while you are waiting at the MUD prompt. MUDs handle prompting and the entering of commands differently than other software you may have used. In fact, some MUDs have no prompts.

LP MUD

LPMUDs handle prompting in the most simple way. They provide a simple > prompt. At the > prompt, you can issue the commands you want, then MUD will execute them. However, if you are not typing anything, it is entirely possible that you will see other messages from the MUD that appear onscreen. When this happens, you do not receive a new prompt. Because you received the prompt earlier, you can issue new commands at any time.

DIKU MUD

Some DikuMUDs use the basic > prompt that LPMUDs use, while others use a more detailed prompt, such as `< 47Hp 88Mn 87Mv >`, which provides up-to-date statistics on your character. DikuMUDs differ from LPMUDs in that their prompts are updated every time you receive information, such as someone in the room says something or you receive a `tell` or message.

MOO MUSH MUCK

MOOs, MUSHes, and MUCKs do not have prompts. You just type and your requests are processed by the MUD. (Sometimes, especially on very crowded MOOs, there will be several seconds of delay between your command and its results.) Occasionally you may even enter several commands in advance of the commands that are being executed.

Navigation: Moving Around on a MUD

For moving from room to room, MUDs use the cardinal directions—north, south, east, and west. Rooms also use up, down, northwest, northeast, southwest, and southeast. Some subset of those 10 directions make up the exits for any given room. Because directions are used so often, you can use abbreviations.

COMMAND

The following is a list of the standard MUD directions and their abbreviations:

down	d
east	e
north	n
northeast	ne
northwest	nw
south	s
southeast	se
southwest	sw
up	u
west	w

These abbreviations are not case sensitive. You can use upper- or lowercase letters.

Although these are the standard directions, rooms are not bound by this small set of directions. Many rooms have different directions, with exits such as portal or nexus, and are explained in the description of the room and listed among the exits. When alternate exits appear, there usually, although not always, is a reason for them. So watch for clues in the room's description, contents, and in neighboring rooms.

NOTE

This chapter has many images of MUD rooms, players, and other MUD components. Because they are taken from an LPMUD, there may be minor differences among them and what you might see on other types of MUDs. The differences primarily are cosmetic, and the examples used here should be generic enough to be valid on most types of MUDs.

The best way to learn how rooms work is to see them. Following is a brief tour of a small section of a MUD:

```
You are on a bridge crossing a frothy river. To the north there appears to be a
small town. Far off in the distant forest you see a gothic spire rising into
the sky.

There are two obvious exits: south and north.
```

NOTE

The fact that the preceding line says `obvious exits` does not necessarily mean that there are hidden exits. There *could* be hidden exits in any room, but they likely would be addressed or hinted at in the room's description. The use of the `obvious exits` format has become standard practice on many LPMUDs. Other MUD types may use other ways to describe exits. Examples of room formats from other types of MUDs are discussed in detail, later in this chapter.

```
> s


A large open plain. There is a river to the east and some kind of building to
the
west.
There are two obvious exits: west and east. The river also winds north and a
bridge crosses it there. A SIGN: URGENT, READ IMMEDIATELY!.

> n
```

NOTE

The sign in the preceding room actually is an object in this room. You can look at it and read it. Watch for objects like this in rooms as you explore on a MUD.

You can look at this sign, which will show further information, revealing the information on it. The `URGENT` message is just a ploy by one of the MUD's wizards to attract players into this area of the MUD and to add a little character. Its message is a plea for help, and in the real world, this plea would be considered important, thus the use of the word `URGENT`.

```
You are on a bridge crossing a frothy river. To the north there appears to be a
small town. Far off in the distant forest you see a gothic spire rising into
the sky.

There are two obvious exits: south and north.

> n
```

As you can see, after going south and then returning north, you have ended up in the same place. This particular area upholds a logical sense of space and reality. Some areas may not.

```
You are in the southern part of a small town. To the west is a strange store.
Little demons keep running out of it carrying small packages. To the east there
is an ancient library. There is more of the town to the north.

There are four obvious exits: south, east, west, and north.

A bulletin board.

> n

You are now in the northern part of the town. To the east is a small trading
post. To the west is a strange building with a large domino over its door. To
the north you can still see the gothic spire breaking into the sky.

There are four obvious exits: east, west, south, and north.
```

```
> n

You are at the edge of the small town. To the north the spire draws your
attention.

The town is back to the south.

There are two obvious exits: south and north.

Azatoth the Master of Dragons (Mortal).

> n
```

In the room just shown, you see Azatoth, a player. This is how players will appear when you see them in a room. You can recognize him as a player (rather than a monster) because of the `(Mortal)` next to his name. Different MUDs will differentiate players and monsters in different ways. Some MUD types, such as MOOs, MUSHes, and MUCKs, usually do not have monsters.

```
The sounds of the town leave you quickly as you find yourself suddenly in a
very dark and gloomy forest. The spire draws you towards it.

There are two obvious exits: south and north.

> n

You are in a very dark and gloomy forest. The spire draws you towards it.

There are two obvious exits: south and east.

An Imp.

> e
```

The `Imp` in the preceding session is a monster. He is just like all of the other objects, but he can hurt you. If you attack him, he will defend himself. Often monsters will start with `A`, `An`, or `The`, as opposed to players that rarely start with this way. But again, it is not always easy to differentiate players from monsters.

```
The forest seems to be thinning out here. To the east you see the rest of what
is attached to the gothic spire. An ancient, huge cathedral stands before you.

The entrance to the cathedral is to the east. The large double doors that used
to mark its entrance have long since decayed. Above the frame of the doors is a
seven rayed star, the symbol of chaos.

There are two obvious exits: west and east.
```

```
> e

You have stepped into the eerie cathedral. Many of the vestments of a church
are strewn about. Most of what has not completely decayed is of a purple or
green hue.

To the east you see a large dias. On it is a huge statue that seems very
menacing in the shadows here. You also hear a frantic chanting that seems to be
coming from the dias.

There are two obvious exits: west and east.

> e

On the dias are several strangely dressed clerics dancing around a swirling
vortex of chaos. You think you could run past them into the vortex without them
being able to stop you.

There are two obvious exits: west and vortex.

A strange looking cleric.

A strange looking cleric.
>
```

As you can see in the preceding room, there is an exit called `vortex`. Although many MUDs use the basic directions for navigation, occasionally there are exceptions. There also are MUDs that do not rely on the basic directions at all.

As you can see from that brief MUD tour, a MUD contains many different components. The key to understanding MUDs is the concept that all the pieces are objects. The sign and the bulletin board you saw in the preceding rooms, for example, are objects. If you want to see what they look like, you can use `look at sign` or `look at board`.

`look` or `l` show your surroundings.

`look at <object>` gives you detailed information about the object. This commonly is called the object's long description.

Some MUD types do not require the `at` and others do. `l` and `lo` often are useful abbreviations for `look`.

How do you know what to call the object? Every object has a name. This name also is known as the object's *ID*. Other players in the MUD have unique IDs—their name. You can use a player's name to talk specifically with him or her or to see if he or she currently is playing. Non-living objects in the game also have IDs. The bulletin board's ID is `board`, so `look at board` and `read board` will allow you to use the object. These IDs also are case insensitive, thus you could use `look at tarod` or `look at Tarod`, even though Tarod is a proper name.

Notice the `read board` command. This command is not an *official* command, meaning that is not part of the MUD itself and will not work in all cases. Rather the `read` command is defined in an object—in this case the bulletin board. With commands like this, the command will only work if you are in the same room as the object or you have the object in your possession. You will find that there are many commands defined in this way. You usually can find out which commands are available on objects by looking at them or by using commands that can be logically deduced, such as `read`ing a book (or bulletin board) or `dig`ging with a shovel. Not all objects have commands associated with them; however, many do. MUDs do not always do a good job of differentiating these objects, but if you think the object is more than decorative, you might try a few commands that relate to the object and see what happens.

Some objects may respond to several names. The `Imp`, for example, may be known as `Imp`, `Monster`, or `Devil`. You usually can determine an object's name from its description or appearance. Generally, when interacting with another object, the name you will use will be one word. The strange looking clerics can be viewed with `look at cleric`, but not `look at strange looking cleric`. This is the case on most MUDs, although some may allow you to use the full name of the object.

This use of IDs is oriented toward LPMUDs and Dikus because the object is created with a single `name`, and then with several aliases, and a full name. Thus, `A strange looking cleric` has the name (or ID) of `cleric`, the full name of `A strange looking cleric`, and possibly an alias such as `monster` or `priest`. MUSHes and MUCKs work slightly differently. For example, `A strange looking cleric` would be matched by `A strange`, `looking`, `cleric`, `A str`, `looking cl`, but not by `nge looking`.

Throughout the course of this book, you will learn many new MUD commands. These commands function much like the `look` command and may require you to use an object's name. So remember, the ID or name of anything you see is likely to be the most obvious word in its description.

Now you know the significance of objects in the game and how they work from a player perspective. The next section details how objects work within the game. Although this is a little more technical, it can help you understand the basic reality of MUDs.

How MUDs Work

When you use a MUD, you are entering a *virtual world*. That world has a structure and framework just like the real world—only the rules are different. MUDs don't start on a base of molecules—they build from a base set of objects. Objects are pieces of computer code that have specific tasks. The objects inside of a MUD have been programmed to interact with each other in various different ways. While the number of possible objects is nearly infinite, we can examine several of the most basic objects.

The *player* Object

The player object is one of the most important pieces of the MUD. When you log in, you are assigned a player object. That player object then copies all the saved information about your character into itself, thus molding to that identity. In fact, often times when you connect to a MUD, you will see `Connecting to obj/player...` or another similar message. This is because on many MUDs, the login process is even a part of the player object; however, on many newer MUDs you will see this message after you log in.

The player object has many different functions that vary widely from MUD to MUD. At the core, however, the player object is your *virtual body*. People can look at you (your player object) and you can look at them by using the `look` command.

What is to follow is taken from an LPMUD and is highly MUD specific. Again, it is explained and shown here as an example of what you might see. If you are standing in a room with several other players and type **`look at tarod`**, you will see the following:

```
Chaos Lord Tarod the Wizard (Wizard) (male) (elf).
Tarod is a tall, gaunt elf with solid black hair and glowing green eyes.
Tarod has a scar on his right leg, his left arm, his right arm, his forehead
and his left cheek.
He is in good shape.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Tarod wears the gray robes of a Mage.
      Tarod is carrying:
torch.
stone cutter sword.
frost sword.
torch (lighted).
Magical Full Plate (any) (worn).
Holy Avenger (wielded).
Magic Shield (any) (worn).
```

The preceding information is collectively known as Tarod's *long description*. Look at the different types of information you have after looking at Tarod.

```
Chaos Lord Tarod the Archmage (Wizard) (male) (elf).
```

The preceding is Tarod's *short description*. The `(male) (elf)` is part of his overall description rather than his short description. The rest of what you see is his short description, which is what you would see if you walked into a room where he was standing, such as the following:

```
The forest seems to be thinning out here. To the east you see the rest
of what is attached to the gothic spire. A towering, ancient cathedral stands
before you.

The entrance to the cathedral to is to the east. The large double doors
that used to mark its entrance have long since decayed. Above the frame
of the doors is a seven rayed star, the symbol of chaos.

There are two obvious exits: west and east.
Chaos Lord Tarod the Archmage (Wizard)
```

To find out whether he is a male elf, you would have to `look` at him. The following is the next piece of information in the long description of Tarod:

```
Tarod is a tall, gaunt elf with solid black hair and glowing green eyes.
```

This is the description that was input when he used the `describe` command.

COMMAND

`describe <description>` adds a description to your `player` object. You may have to work with this for a little while to achieve the results you want. Look at yourself (`look at <your name>`) to see the results. Some MUDs insert your name at the beginning of what you type for `<description>`, and others do not.

The way descriptions are set can vary widely between MUDs. The preceding syntax is for LPMUDs. DikuMUDs have a special set of login options, one of which allows you to set your character's description.

On a MOO, you would use `@describe me as <description>`, and on MUSHes and MUCKs you would use `@desc me=<description>`.

This particular MUD inserts the player's name and adds a period to the end of the description that the user enters. This type of description almost always is optional and only serves cosmetic purposes; however, it can differentiate your MUD character and project his or her MUD persona.

```
Tarod has a scar on his right leg, his left arm, his right arm, his forehead,
and his left cheek.

He is in good shape.
```

The scars indicate the number of times Tarod has died. On LPMUDs, every time you die you get a scar. The way in which scars are assigned varies widely among MUDs and generally is only cosmetic. The second line indicates that Tarod is in good shape. If Tarod had just run out of a fight with a powerful monster, this line might instead be one of the following (each of which indicates a different level of damage has been inflicted):

```
He is not in good shape.
He is hurt.
He is in bad shape.
He is in very bad shape.
```

TIP

On combat MUDs, these messages can be very useful. You can get a good idea as to how close a monster is to dying by looking at them. `He is in very bad shape.` usually is a good indicator that a few more hits will kill it.

The next line indicates that Tarod is in the Mage's guild (guilds sometimes are referred to as classes). This is entirely dependent on the particular MUD—some MUDs do not have guilds or classes. Although this line may have no specific relevance, it is important to know there may be some variation among the long descriptions you see on different MUDs.

```
Tarod wears the gray robes of a Mage.
```

Finally, you have reached the last part of the long description.

```
Tarod is carrying:
torch.
stone cutter sword.
frost sword.
torch (lighted).
Magical Full Plate (any) (worn).
Holy Avenger (wielded).
Magic Shield (any) (worn).
```

The preceding shows Tarod's inventory. The `player` object is a container for other objects—it can hold them and use them as necessary. To see what you are carrying, use the `inventory` command.

COMMAND

`inventory` or `i` gives you a list of the items you currently are carrying.

Notice that some items have next to them additional information in parentheses. The `(any)` next to the `Magical Full Plate` and `Magic Shield` means that any size character can wear that particular piece of armor. This MUD has several sizes of players, so some armor is `small`, `medium`, or `large`. Dwarves, for example, can only wear armor that is for size `small` or for any size.

Not all MUDs use armor sizes. The second piece of information, the `(worn)` next to the `Magical Full Plate`, simply means that Tarod is wearing that piece of armor. `(wielded)` means that Tarod is wielding the Holy Avenger as his weapon. On non-combat MUDs, `wielding` and `wearing` usually are not an issue except perhaps as an aesthetic addition.

There is one last concept that is important in the `player` object, although it is primarily used on combat MUDs. Inside the game, the `player` object is alive. The `player` object is living and has a heartbeat. The heartbeat is not like ours; it doesn't pump blood, and if the heartbeat stops, it doesn't mean you are dead. In fact, it is kind of like a heart attack—if you get any messages that indicate you do not have a heartbeat, you should contact a wizard to correct the problems.

The concept of a living `player` object is very important in a combat MUD because there would be no challenge or risk if you could not die. If you run out of life (by whatever measure the particular MUD uses—usually hit points) you will die. Death can mean a variety of different things, depending on the MUD you are on. On combat MUDs, however, death is bad.

The `player` object is not the only living thing on most MUDs; usually monsters also exist. These monsters are responsible for most player deaths. Players sometimes are allowed to kill each other as well, but that is a whole new can of worms that is addressed later.

The *room* Object

The second most important object on any MUD is the `room` object. The fabric of the MUD is woven from various derivatives of the room object, and, once they are sewn together, the end result is the virtual world. As you move about the MUD, you actually are being transferred from room to room. Each room has been configured in a different way and with a different appearance; but at the lowest level, they are all the same. The basics of a room are its appearance, exits, and contents. An empty room looks like the following:

```
> l
Village
     This is a small little mountain village nestled in
     heart of the Bridger mountain range. The mountains are
     huge and breathtaking. Off to the east lies a pub and
     off to the west lies the bank.

    There are four obvious exits: north, south, east, and west.
>
```

The preceding room follows the standard procedure of listing the exits at the bottom of the room's description. Also, rooms often detail the exits as part of the room's actual description. A crowded room (full of players) looks a little different, as in the following example:

```
> l
You are on the outskirts of the town. Short roads lead off to the south
and north. More shops can be seen to the east, and forest to the west.
    There are four obvious exits: north, south, west, and east.
Backlash the Squire (Mortal).
Maquis the Squire (Mortal).
Enter the Master of the South Wind (Mortal).
Thalassa Mistress of the Outer Provinces (Mortal).
Black Hawk Naberius son of Galahad and Eliatra (Mortal).
A horse drawn Carriage waits here.
>
```

On LPMUDs and DikuMUDs, only wizards can build rooms. On other MUDs (especially MOOs), a broader segment of the MUD population can build rooms. In the following text, the term wizard means anyone who can build rooms within a MUD. Wizards configure the first two parts of a room—appearance and exits. Sometimes wizards also configure rooms to automatically load certain objects or monsters. The *appearance* is just a descriptive statement that a wizard has decided is appropriate for this room, depending on its purposes. Rooms in the same area usually are consistent and contain some type of theme.

Exits also are defined by wizards, so it's possible for weird things to happen. Most MUDs have no internal consistency checks, so it's possible that if you go east from a room 1 into room 2, and then go west from room 2, you won't end up back in room 1. These types of problems are weeded out during testing, but if you run into a paradox like this, report it to a wizard.

Finally you are to the contents of a room—the most important part. If you are standing in a room, you are one of its contents. Most of the time, however, the computer is clever enough not to show you as one of the inhabitants of the room (although I've had this happen before).

```
> l
The forest closes around you further still.
The air becomes moist and stifling...
The darkness till surrounds you!
    There are two obvious exits: east and west.
A caru antler.
A Map.
Scroll of Identify.
Wand of Magic Missiles.
A Glowing orb.
```

```
Demonwhip.
Demon Shield (any).
Gothmog's ring (any).
```

In the preceding example, you see a room full of inanimate objects. In previous examples, you have seen rooms that contain other players or monsters. You learn in the following sections how to recognize different objects.

Room Variations

Other types of MUDs arrange rooms in different manners. Included here for reference are rooms from the other types of MUDs (excluding LPMUDs, which have been used as previous examples).

A Room from LambdaMOO

The following is an example of a room from LambdaMOO:

```
The Living Room
It is very bright, open, and airy here, with large plate-glass windows looking
southward over the pool to the gardens beyond. On the north wall, there is a
rough stonework fireplace. The east and west walls are almost completely
covered
with large, well-stocked bookcases. An exit in the northwest corner leads to
the
kitchen and, in a more northerly direction, to the entrance hall. The door into
the coat closet is at the north end of the east wall, and at the south end is
a sliding glass door leading out onto a wooden deck. There are two sets of
couches,
one clustered around the fireplace and one with a view out the windows.
You see README for New MOOers, Welcome Poster, a fireplace, Cockatoo, The
Birthday
Machine, Helpful Person Finder, and lag meter here.
Lalysa, Yellow_Guest, Forever_Guest, LAme-o (avoid disappointment - lower your
expectations), Cornelius, Cyber_Rogue (distracted), Bloodthorn, Pink_Guest,
Lecturer (a hard but fair marker.), and Quicksand are here.
```

A quick dissection of this room gives you three parts. The first is the description of the room, which in this case also provides a clear description of the exits from the room. The `You see README ... and lag meter here.` portion lists the inanimate objects in the room. And finally, `Lalysa, ... and Quicksand are here.` lists the players currently in the room.

A Room from the Elite DikuMUD

DIKU MUD

The following is an example of a room from a DikuMUD:

```
The Temple Of Midgaard
   You are in the southern end of the temple hall in the Temple of Midgaard.
The temple has been constructed from giant marble blocks, eternal in
appearance, and most of the walls are covered by ancient wall paintings
picturing Gods, Giants and peasants. Large steps lead down through the grand
temple gate, descending the huge mound upon which the temple is built and ends
on the temple square below. To the west, you see the Reading Room. The donation
room is in a small alcove to your east.
An automatic teller machine has been installed in the wall here.
+Clara the Fairy Dark-Blade (linkless) is standing here.
Jahafir - Disciple of the Wolf God is sleeping here (Zzzzzzz).
Adorable Odie the Minotaur Overlord/Overlord is standing here.
Tyr the Half-elven Sentry is standing here.
+The Priest is standing here, offering his services.
[Exits:neswd]
```

This room has a few more pieces. It begins with a short description of the room (`The Temple of Midgard`) and then goes into the detailed, long description, which gives information on the available exits. It then lists the animate and inanimate objects in the room. The characters with `is standing here` or `is sleeping here` next to their names are players. The rest of the objects are either inanimate or monsters. Finally, the room's description ends with a list of the exits.

A Room from the Dark Gift MUSH

MUSH

The following is an example taken from a MUSH.

```
Grant Street - 100 Block Downtown(#0RHJMhs)

Grant Street and Liberty Avenue Intersection

        Presiding over the assemblage of glass, marble and steel that are the
stoically upper class buildings populating this oddly triangular intersection
is
the massive bulk of the Amtrak rail station, it's somber grey stones holding
aloft a sloping dome of a roof. Traffic, both foot and otherwise, runs north
and
south along the cobblestones of Grant St. here, always under the watchful eyes
of
the occasional stone gargoyle or angel adorning this office complex or that. If
you were to follow Grant far enough north through the urban canyon, you would
find the Greyhound bus station, but not before passing the elegant Vista
International
Hotel, posh and stately amongst the more modern structures here, no steel
to be found on its aged form, only stone and marble; the remnant of a different
time. Intruding upon the harshness and refinement of Grant is the asphalt of
```

```
Liberty Avenue, a younger tributary of the cobbled road, leading to areas far
seedier
than this.
You arrive at the intersection of Grant and Liberty, a busy intersection in
Down
town Pittsburgh.
Contents:
Kelby
Tomanelle
Teresa
Logan
Obvious exits:
Parking Structure (PS)  Subway <SUB>  Pittsburgh Employment Office (PEO)
East <E>  Southwest <SW>  South <S>
```

This MUSH uses a system similar to the LPMUD examples, except that it specifically separates the contents of the room with the `Contents:` marker. Everything from `Contents:` to `Obvious Exits:` is a player.

A Room on FurryMUCK

The following is an example of a room on a FurryMUCK:

```
[Newcomers, type 'behind' to get to a visitor's center for help adjusting to
FurryMuck. You should find yourself another home as soon as possible; type
'@link
me = #4498' to set your home to the Unicorn Inn, a temporary sleeping area.]
West Corner of The Park
This corner of the Park has a few shrubs and bushes. The main feature is a
huge wooden bandstand, painted white, and surrounded by a roughly fan-shaped
arrangement of wooden folding chairs for furries to sit upon while listening
to the band or just chatting. The lawn slopes down to the shore of the pond,
which is to the east.
The Park spreads to the north and south. There are trees to either of those
directions, the southern part having thicker woods with a narrow trail
meandering into them. To the west, you can see pavement and make out some
movement; you are looking at Cougar Boulevard. You can also see a narrow gap
between the side of the bandstand and the ground, letting onto darkness.
Contents:
Johnny
Gentaur
Dakka
MacPhisto
J.C.
Farin
Raster
Keyth
Hremp
Gaoth
Nightfall
Akaba
Ashtoreth
```

```
Opal
Sandy.Claws
Silhouette
Lumpy
Bulletin Board
```

This MUCK details its exits in the body of the room description. It then groups all the objects in the room under `Contents:`. Most of the items listed here are players, except for `Bulletin Board`. It should be obvious which objects are players and which are not. If you cannot tell, `page` the object and if it responds, it probably is a player.

Other Objects

MUDs and MOOs also contain objects that serve no real purpose. They do not uphold the reality of the game, they do not house your virtual alter ego or someone else's, and they don't pose a threat to your character. They are there purely for aesthetics and fun (or in the case of combat MUDs, sometimes as an alternate form of money). The following is an example of cosmetic object that is fun to play with, but serves no purpose other than entertainment.

```
> i
A magic ball.
> look at ball
An offical magic ball! throw <player> or kick <player>.
> throw striker
You throw the magic ball to Striker.
> smile
You smile happily.
You see a magic ball flying through the air.
The magic ball is swiftly thrown to Guest by Striker.
Guest jumps into the air and catches it.
You catch a perfect spiral thrown to you.
```

As you can see in the preceding example, the `magic ball` is a fun toy. Everyone else in the room saw a character jump up and catch the ball when it was thrown back. It is possible to throw the magic ball to anyone, anywhere on the MUD—there are no geographical boundaries. The magic ball has no value—it can't be sold and it can't harm anything—it is purely for fun. If you try to sell the ball, it will explode (not hurting any bystanders, but ruining the ball).

Many other objects exist in the many MUD worlds, but they are diverse and there is no set standard of objects. You will find that many non-standard objects have funny idiosyncrasies, but most things work the way you would expect. If you find a torch, for example, you probably will be able to light the torch and create light. Remember, you may have to fiddle around to find the commands to use certain items, but often they are shown in the item's long description (what you see when you look at the item). The magic ball, for example, gives its usage instructions in the long description.

Here are some examples of commands you can use to manipulate basic objects.

```
> look at sapphire
A huge star sapphire.
> give sapphire to dartagnan
Ok.
Dartagnan smiles happily.
```

COMMAND

`give <object> to <player>` gives to a player in the same room with you the object in your possession. In the preceding example, the character gives a sapphire to Dartagnan.

```
Dartagnan gives sapphire to Tarod.
> drop sapphire
Ok.
> l
A large open plain. There is some kind of building to the east.
    There are four obvious exits: north, south, east, and west.
A Star Sapphire.
Dartagnan the Master Thief (Mortal).
> smirk
You smirk.
Dartagnan takes: A Star Sapphire.
```

COMMAND

`drop <object>` drops an object you have in your possession. The object will be in the room you currently are occupying.

```
Dartagnan gives sapphire to Rosa.
Dartagnan leaves down.
Rosa drops the sapphire.
> get sapphire
Ok.
```

COMMAND

`get <object>` gets an object from the room you are in and puts that object in your inventory (or your possession).

Special Items on Combat MUDs

Weapons and armor are critical parts of any combat MUD. If your character is not well-equipped with the right supplies, he or she will die quickly. This section does not discuss weapons in detail, but will touch on what they look like and how they work. Weapons and armor are discussed in more detail in the chapters specifically relating to LPMUDs and DikuMUDs.

```
Dartagnan takes sword from bag.
Dartagnan gives sword to Tarod.
> i
Black catsword.
An old short sword.
> look at sword
This sword looks evil.
The sword is gold and black. You can see a
few runes imprinted in dried blood.
> look at sword 2
A rusty, old sword that looks like it might shatter at any time.
> say Your name is from one of the three musketeers, right?
You say, 'your name is from one of the three musketeers, right?'
 Dartagnan nods.
> wield sword
You wield black catsword.
Ok.
```

COMMAND `wield` *<weapon>* enables you to wield a weapon that currently is in your possession. If you get into a fight, you will be using the weapon (and any extra power it gives you) rather than your hands.

```
> i
Black catsword (wielded).
An old short sword.
```

As you can see, the weapon now is marked as being `wielded` when this character does an inventory. Also notice that when you look at other characters, you can see the weapons they are wielding. Some MUDs may allow wielding in the right hand, left hand, or both (this likely will be explained somewhere on the MUD). Don't be surprised if you see `(wielded in right hand)` after someone's weapon.

```
> give sword to dartagnan
You must unwield your weapon first.
> unwield sword
You unwield your weapon.
Dartagnan removes armor.
Dartagnan gives armor to Tarod.
> give sword to dartagnan
Ok.
> look at armor
This suit belonged to Ellefson, the protector of Mustaine.
It is very sturdy. It also emits light.
> give armor to dartagnan
Dartagnan wears glowing platemail.
Dartagnan gives armor to Tarod.
> wear armor
Ok.
```

COMMAND

wear *<weapon>* enables you to wear armor that currently is in your possession. If you get into a fight, the armor will provide an added level of protection beyond what you normally might have.

```
> i
Ornate Elven Chainmail (medium) (worn).
```

When you wear armor, it, like weapons, is clearly noted for all to see. MUDs also have different types of armor, such as boots, gloves, helmets, amulets, rings, and so on. You will only be allowed to wear one of each of these types of armor. The different categories of armor vary among MUDs. On this particular MUD, the armor is followed by `(medium)`, which denotes the size of the armor.

NOTE

A dwarf cannot wear `medium` armor.

```
> look at armor
This is a nicely forged suit of Elven chain mail.
```

Summary

Because the world of MUDs is wide and varied, you probably will find a MUD (or several) that you might want to call "home." This chapter has explained the varied types of MUDs. Remember that much of this chapter uses LPMUDs as a working base, and addresses the many differences you will encounter when traveling to DikuMUDs, MOOs, MUSHes, and MUCKs. These differences are further explored throughout this book, as each of these MUDs has a chapter devoted to it.

I hope you have gained a feel for MUDs, getting the gist of how they work so that you can better deal with the diversity when you encounter it. As you continue, you can focus on sections of this book that relate to the type(s) of MUDs that you find most interesting. Next, Chapter 3 shows you how to actually connect to a MUD.

3

CHAPTER

Your MUD Persona and Role-Playing

Part of the allure of MUDding is the capability it gives players to indulge in the use of an alter ego that can be created on a whim or deeply developed and thought out. This alter ego is the character a player uses in the MUD world. Now, sometimes it is entirely possible, and there are some MUDs that have this focus, that there is no character, only a player. On these MUDs, it should be immediately obvious that everyone is just himself or herself. On these types of MUDs, people usually use their first name or a nickname. Because these MUDs basically are an advanced chat system, they are not addressed in this chapter.

This chapter instead discusses the ways in which you can develop a MUD character for one of the MUDs that has a virtual world populated by characters that are the alter egos of human players. Now, there is nothing that says this character will not have all or some of the player's personality; but when the character's personality is the same as the player's, it still is different from a MUD without characters. So now let us go forth and discuss this idea of a MUD persona.

Your MUD Persona

Most MUDs focus on some sort of role-playing. For a new user, this may be the most difficult part of MUDing. Most people who play MUDs choose a name they will use on MUDs (or sometimes just one specific MUD). This pseudonym is called a *MUDname*. MUDnames often come from the user's favorite books, movies, or TV shows, or are made up. The MUDname usually becomes the genesis for a new persona. Your MUD persona may act completely differently from you, which is the basis for role-playing. As you play a MUD character, over time you will begin to develop that character in different ways. This might mean that you just project your RL (real life) personality into the MUD world or that you develop a completely new personality for your MUD character.

So how do you choose your MUDname and persona? On MUDs, I use *Tarod*, who is a fictional character from a series of books by Louise Cooper. *The Initiate*, *The Outcast*, and *The Master* tell of Tarod's development from a boy to his ascension to godhood as one of the seven lords of chaos. I set my MUD character's description to mirror the fictional Tarod and then set about developing this character. I also let a lot of my real life personality seep into this new MUD persona, thus, it is a subtle blend of my personality and a fictional character.

I have seen MUDnames come from science fiction and fantasy books, mythology, religion, rock stars, real names, and the imagination. Where you get your MUDname is a personal choice, but consider it carefully because the name you choose can affect your MUDlife. An interesting MUDname can attract unwanted attention. A final note on MUDnames—keep in mind that just because you take your name from a fictional character or a mythical being, you don't have to adopt the personality of that character.

Following are a couple of other examples of MUDnames and where they are from:

Bleys	Frank Stevenson, who is the author of Chapter 13, takes his name from a character in the *Princes of Amber* series of books by Roger Zelzany.
Moira	Jennifer Smith, the technical editor of this book, takes her name from Greek mythology. Her name is the singular word for the Moirae, the Fates. In the singular, it refers to the overall randomness of the universe.

TIP

A popular MUD pick-up line is "That's a cool name, where did it come from?"

The most important thing to remember when on a MUD is that anyone you interact with might be role-playing. You could spend hours talking to folks about different things and in reality, they have no interest or have completely different personas. This is part of the allure of MUDs; however, it also is a potential problem. While you can create the outgoing, self-confident image you have always wanted, or be an evil, uncaring tyrant, you have to be careful because occasionally (and I know, you'll say this can never happen to you) you become very close to people you meet on a MUD. Once you become close, it's important to make sure that the other people are really being themselves and not fictional MUD characters. If you aren't sure, ask. If you are only on the MUD for the game aspects, make it clear to anyone who asks you personal questions that you are playing a character and don't want to talk about personal things online. Don't be surprised if those with whom you have spent months killing monsters suddenly become quiet if you ask them for their real names.

Gender Relations

On the Internet, the majority of users are male. Although this rapidly is changing as the Internet grows and becomes more mainstream, it still is the case as of the writing of this book. This imbalance holds true on MUDs as well. A MUD with 50 users logged on most likely will have about 42 male characters and 8 females. Because of this skewed ratio between the two sexes, female characters tend to be showered with attention. New female characters on gaming MUDs almost always are offered gold, powerful items, and help in killing creatures.

Playing a female character certainly will make it much easier to get help from other players, but it also has its drawbacks. Female characters tend to become magnets for every uncouth male player on the MUD. This can lead to lewd comments, leering, and so on, which can be very offensive. Be aware that MUDs tend to have the same problems as the real world, and sexual harassment and rude behavior appear in virtual worlds as well. If you are harassed or someone is being rude (such as making obnoxious innuendoes, "kissing" you, and so on), ask them to stop. If they persist, complain to one of the administrators. Most administrators will not tolerate this type of behavior and will appropriately discipline offenders.

Another twist in all of this is *gender bending*. Because there are no rules for choosing your MUDname and persona, many people may do things that surprise you. It is not uncommon for men to create female characters—because of the benefits. Women sometimes create male characters in order to avoid some of the potential problems of playing a female character.

Role-Playing

When you enter a MUD world and assume your MUD alter ego, you no longer are yourself. You are an actor in a scene with many other actors and actresses. Your role, however, is not clearly defined by a script and you have complete artistic control. As you play out the scene, you inevitably will interface with other actors and react to the roles they play.

The role you have chosen to play might be your own personality, it may be based on a fictional character whose name you are using, or perhaps it is a completely new alter ego that is not based on anything in particular. The more you use this role, the more this new alter ego will grow and evolve. It may take on more of your real life personality traits or it may become more unique as you allow yourself to be immersed in this fictional world.

A good example of role-playing on non-combat MUDs is FurryMUCK. *FurryMUCK* is a very popular MUCK that has been around for quite some time and has been mentioned in several magazine articles about MUDs. On FurryMUCK, all the characters are furries. "What is a furry?" you ask. A *furry* is an anthropomorphic animal—a creature that looks like an animal but is intelligent and has human characteristics, such as Bugs Bunny and Daffy Duck. Players on FurryMUCK assume the role of an animal and play the part. You might see these anthropomorphic animals playing games with their tails or making chipmunk noises (rather than just chatting). The fact that you are playing an anthropomorphic animal gives you a lot more freedom in the way that you can express yourself (cats can purr, dogs can bark, and so on).

The idea of acting a part in this new virtual world may sound odd at first, and is not required—many players simply imbue the character they create with their own real life personality. Acting a part, however, is a fundamental step in understanding MUDs, especially combat MUDs. The most important distinction is that the different types of MUDs focus on two different (although not mutually exclusive) forms of role-playing. The first is pure role-playing, which has been discussed in the last few paragraphs. This means assuming a role and acting it out. You interface with others, react to what they do, and have fun playing a character. The other form of role-playing is more game oriented and is detailed in the following section. It involves creating a character, imbuing that character with specific abilities, and then developing those abilities.

Player or Character?

You, as a person, are a player on the MUD. The visual being that you have created (your MUD alter ego) is a character. Many times on MUDs, these two are confused, and often, they run together. In this book, I have tried to keep them separated, but it can become pretty confusing. So I have devoted some effort to keeping them distinct in parts of this chapter that discuss what happens when a character dies. Remember, what usually happens on a MUD is happening to your character and not you. (Well, unless you are talking about real life, and then you are you and not a character.)

A player character (PC) is a character played by a real person. There also are non-player characters (NPC) that are either played by wizards or the MUD's computer. These NPCs usually are monsters (although some people only consider very advanced, semi-intelligent monsters to be NPCs and the rest to be drones) or wizards playing NPCs. PCs are other real people. And it is not always easy to tell the difference.

Another important difference is OOC and IC. *OOC* (*Out Of Character*) means that you are talking as you, the player, and not as your character. You use OOC to talk about real life, ask for help on using the MUD (your character doesn't know about the MUD), or just basic real world talk. Some MUDs even have special OOC rooms for direct player-to-player talk. Remember, anything designated as OOC will be taken as something said player-to-player in real life. `tells` and `pages` also are generally assumed to be OOC.

On the other hand, *IC* (*In Character*) is used on many non-combat, role-playing MUDs to indicate that you are returning to your character after an OOC discussion. It means you are playing your character and anything you do is coming from your character and not you, the player. On role-playing MUDs, it is assumed that you are IC unless you indicate otherwise or ask some blatantly OOC question.

Before continuing on with the discussion of some of the systems that combat MUDs use to define characters (specifically to allow the MUD to manage combat between players and monsters or other players), let me address another type of MUD.

MUSH

There are MUSHes on the Internet that use statistics, like other combat MUDs, and probably even fall under the classification of combat MUD. On these MUSHes, however, you don't gain experience from fighting and fighting is pretty rare. You do have statistics and other characteristics similar to the ones you are to learn about, but you do not use them exclusively for combat. You role-play a character who is an active participant in the world. Many of these MUSHes are based in the *World of Darkness*, a gothic version of today's society that also is occupied by vampires, werewolves, and mages. This world and its accompanying game system are based on a popular and more traditional role-playing game published by White Wolf Game Studio.

On these MUSHes, your character will have stats and the other characteristics about to be discussed, but if combat arises or you need to use your special skills in some way, you call a judge. A judge is a special type of administrator (like a wizard) who can arbitrate combat or other special actions by characters. You also can have judges construct objects for you if they fight for the needs of your character. These particular MUSHes focus on role-playing and players vote to give experience points to players who do a good job of role-playing their characters. If you think you will be playing this type of MUSH rather than just the non-combat MUSHes, you should read on. If you expect to only be playing on social MUDs, you may want skip to Chapter 4.

LP MUD

Most of what you will read in the rest of this chapter is only relevant to combat MUDs.

The Statistical Being

You have this new character you have created on a MUD. You want him or her to be able to do things, such as kill monsters, buy items, and move around. How do you do that? How are your character's capabilities defined? This is the core difference between a combat MUD and a social MUD. On *social* MUDs, there are no intrinsic differences between characters (other than perhaps that of player and wizard), whereas on *combat* MUDs, characters may be very different from one another.

score outputs the vital characteristics that make up your MUD character. These statistics and numbers are the lifeblood of your MUD character and determine his or her capabilities in combat, spellcasting, and other MUD activities.

To begin looking at the non-social aspects of a MUD character, or the "statistical being," look at the following use of the score command:

```
> score
 Tyla Initiate of the 6th Circle
-------------------------------------------------
Level: 12
You have 86% of the experience needed for the next level.
Hp: 113/145      Sp:132/132
Money: 22497  (45000)
Guild: druid
Race: elf
Size: medium
Alignment: evil
Str: 8 Int: 8 Wis: 6
Dex: 4 Con: 1 Chr: 4
Wimpy 30%
Hunted by: Nothing at all, aren't you lucky?
You are normal. (Yeah, right)
You are a Druid, protectorate of the races.
To see your abilities, type 'druid'
Ok.
```

The preceding output from score details this character's stats in the form of level, experience, HP (or hit points), SP (or spell points), money (or gold), guild (or class), race, size, alignment, and stats, such as str (strength), int (intelligence), wis (wisdom), dex (dexterity), con (constitution), and chr (charisma). These are the statistics that you will see on many MUDs, but they will vary from MUD to MUD. The chapters on the specific types of MUDs (Chapter 8 for LPMUDs and Chapter 9 for DikuMUDs) go into more detail on the specific types of stats you might see on these MUDs. Further, there also are MUSHes that have stats based on other game systems. For example, there is a large group of MUSHes that use the role-playing game called *Vampire: The Masquerade* as their basis, and hence, use the stats and other characteristics set forth in that system.

At the core of a character on a combat MUD are his or her base statistics (called *stats*). These stats usually are strength, dexterity or agility, constitution or stamina, intelligence, wisdom, and charisma. The stats used vary widely, but the preceding six tend to appear a vast majority of the time. They have become standard because they are used in the *Dungeons & Dragons* games from TSR, Inc. Because many of the role-playing games (both computer-based and otherwise) have been inspired by D&D, many of the core elements are similar.

Recommended Reading on Role-Playing

Advanced Dungeons & Dragons Player's Handbook from TSR, Inc., is the classic book on role-playing and provides detailed information on statistics, races, classes, and more. This book is useful for learning more about the statistics and systems used, such as hit points and levels, and the actual mechanics of combat. Many MUDs are loosely based on the systems described in this book. *AD&D* tends to focus on the gaming aspects of role-playing.

Vampire: The Masquerade from White Wolf Game Studio is a more recent role-playing system. This book does an excellent job of explaining the concept of role-playing with less of an emphasis on the game aspects. For a better understanding of how to role-play a character and develop it, this is the best book I have encountered. Besides its usefulness as a guide to role-playing, there are also several MUSHes based on this game system. (These are the *World of Darkness* MUSHes I discussed briefly earlier in this chapter.)

These core stats can affect your character in many ways. Strength means you can hit harder, dexterity means you can dodge attacks better, constitution means you can survive longer, intelligence means you can cast more powerful spells, and so on. Although the manifestation of these effects may vary from MUD to MUD, the general idea is the same.

Another important part of your character is *hit points*. Hit points (HP) determine your ability to take damage. When you get to zero hit points you generally die. On MUDs there are usually many ways you can get hurt (or take damage)—that is lose hit points—and there also are many ways to be healed—have your hit points restored. Generally, your character will have some maximum number of hit points and a current number of hit points. When these two numbers are the same you are at full health. But if you attack a monster you may get hurt and your current hit points will go down. You need to watch your current hit points closely so that you know if you are in bad shape.

The maximum number of hit points usually is determined by your *level*, your *class*, and your constitution (one of the statistics discussed previously). The effects of your level and class will be discussed soon. Your maximum hit points will not go up often; only when your level or your constitution increase.

Another number, *spell points* (or SP), determines your magical power; although on many MUDs, spell points affect more than your ability to cast magical spells. If your character is a class that can use magic (see classes in the next section), he or she can use spell points to power spells. Other characters may find items that require spell points to become activated, such as magic wands or special swords. Finally, on many LPMUDs, spell points are used for some social commands, such as `tell` and `shout`.

```
tell <player> <message>
page <player> <message>
page <player>=<message>
```

These commands relay your message to the designated player wherever on the MUD he or she may be. The designated player can be standing in the room with you or at the other end of the MUD.

Some MUDs, LPMUDs, and DikuMUDs tend to enable this type of command more than others, and have a command that allows you to send a message to everyone who is currently using the MUD. `shout <message>` gives your message to everyone on the MUD. DikuMUDs have several versions of this. For more information, see Chapter 9.

`tell` and `shout` use spell points on an LPMUD. This partially is to avoid shouts cluttering the airwaves with irrelevant information (requiring spell points severely limits the number of times any one person can shout). `tells` require spell points to keep people from collaborating too closely without being in proximity. LPMUDs emulate fantasy worlds, and a `tell` allows telepathic communications among the players, so this is restricted to maintain game balance. This restriction also helps keep players from abusing `tells` by using them to kill other characters. If there were no spell point requirements, a character could send many consecutive `tells` to another character while he or she is in combat, and the barrage of `tells` could cause the commands the player types to be delayed just enough to result in his or her death from the monster he or she is fighting.

Like hit points, your maximum number of spell points usually is determined by your level, your class, your intelligence, and, depending on the MUD, your wisdom. Like hit points, spell points do not increase very often.

Your character also has experiences. These experiences are quantified in *experience points* (*XP*, *EP*, or *eeps*). You can use experience points in several different ways (depending on the MUD). The primary effect of experience points is that they determine your level. Some MUDs also allow you to spend experience points to increase your stats, skills, and other capabilities.

NOTE

RealmsMUD—from which many of the examples in this book come—makes it difficult to tell the exact number of experience points a character has. On RealmsMUD, the `score` command tells you only the percentage of the way you are to your next level, but not the exact number of experience points your character has. This generally is not the case and usually the `score` command will tell you exactly how many experience points you have. So, on most MUDs, you know the exact amount of experience your character has at any given time and how many experience points it takes to advance to the next level.

Level

The *level* of your character is an indicator of your relative strength. You begin at the first level and advance as you gain experience points. As you advance levels, you will gain new skills, more hit points and spell points, and become more powerful. On many MUDs, there are areas that are restricted to various levels — newbie areas have areas in which only new users can enter, and high-level areas are areas in which only experienced players can enter. Some weapons and armor also have level restrictions.

Your level also may affect your ability to raise your stats. It may even affect your ability to ally with other players. As far as statistics go, it probably is the most important part of your character.

Your Virtual Body

All of the stats and other characteristics that are being discussed here are held by the computer. You have no power to change them except through the game. (Wizards can change them if something goes wrong…or, in some cases, if you bribe them.) But in general, this part of your character will only change as you play. As you gain levels and experience points, you will have opportunities to increase your stats. As you fight monsters, your alignment probably will change. Sometimes, spells can temporarily raise (or lower) your stats or change your alignment.

Anytime you kill a monster, you will gain experience points, which are automatically added to your character. The next time you use the `score` command, you will see that your experience point total has gone up. You do not need to worry about these numbers because the computer (on which the MUD is running) keeps track of them. If you have the opportunity to increase your stats or improve your character in some way, the MUD will prompt you with what you need to do.

When you kill a monster or advance a level, you should use the `save` command. This command saves the latest version of your character so that in case the MUD crashes, you still will have the latest version of your character. (Occasionally the MUD will go down for no reason and everyone will have to log back in—this is known as a *crash*.) Most MUDs have an autosave feature that saves your character on a regular basis, but it never hurts to be safe. When you quit, your character is saved automatically.

The stats that make up your character are almost always yours alone. Other characters cannot see your stats, although they usually can see your level, class, and alignment. Only you can see your stats, hit points, spell points, and experience points. The exception to this rule are wizards because they can look at almost anything on the MUD.

The Class or Guild

Finally, your character will have a *class* or *guild*. This is your character's profession or specialty. As usually is the case, the powers and names of these classes will vary widely among MUDs.

LP MUD

LPMUDs almost always start off new characters as members of the Adventurers' Guild. This class has a few spells (that are gained as you advance in levels) and fight reasonably well. Members of this class have an average number of hit points and spell points. Adventurers are considered to be a fairly weak class and no one tends to remain an adventurer for very long. Adventurers are important because they are standard throughout LPMUDs (although some have done away with them). Also, not all LPMUDs have a rich system of classes or guilds, and on those MUDs, *all* characters are adventurers—there are no other choices.

Guilds are strong in either spells, combat, or special skills. Spells can vary from the capability to teleport to another player on the MUD to a fireball that inflicts damage on all the monsters in the room. Combat is the capability to use weapons or even one's hands to inflict more damage on an opponent and to avoid or better handle damage that is done to you.

Special skills is more of a broad category. Thieves tend to have many special skills, such as backstab. *Backstab* allows the thief to inflict large amounts of damage on an unsuspecting victim. Special skills have the highest degree of variance among MUDs.

Following, as a point of reference, is the list of classes from RealmsMUD. As is always the case, these classes are not standard, but the classes on Realms run fairly close to the standard set on many LPMUDs. After each description, a little insight is offered into what the classes/guilds can really do.

```
REALMS Class Information (read sign).
> read sign
```

Bards

Bards have many special social powers. They can sing songs, shout more, and cause mischief. Most of these capabilities aren't particularly helpful in combat, but they are great for entertainment and role-playing. Bards tend to be able to use almost all weapons and armor. They are average fighters and have a fair selection of spells.

```
            The Bard Guild
The bards are a merry folk. Their main powers are gained through
song. Their various songs give them magical powers, though they
are still formidable fighters. They incorporate many different abilities
of the other guilds, and have a few spells, or songs, of their own.
```

Druids

Druids have average fighting skills and spell casting. They have both offensive and defensive spells and healing spells. They can use any weapons and all non-metal armor. Because druids are neutral, they can use things that are geared toward both good and evil characters. (Druids are a more specialized guild and their implementation will vary quite a bit.)

```
            The Druid Guild
The druids are people of the forest. They live in harmony with
nature, and pay careful attention to their alignment. Due
to their closeness with Mother Earth, they are not allowed
to wear metal armors. Their fighting skills are indeed
formidable, as well as their magical prowess.
```

Fighters

Fighters are, well, the best fighters. They are attacked in the same time period as other classes and can use all weapons and armor, but they don't have magical capabilities. On RealmsMUD, they have a few special capabilities beyond good fighting skills. Fighters also tend to have more hit points than other classes.

```
           The Fighters' Guild
The fighters' guild has two parts: Barbarians and Paladins.
The Barbarians are those who view the barbaric way of fighting as the way of
life, while the Paladins look to the Gods of Law for their fighting strength.
The two groups are based on alignment, with the Paladins being good and the
Barbarians being evil.
```

Mages

Mages are the masters of spellcrafting. They have spells for virtually ever occasion, except healing. Mages cannot use most weapons; they are restricted to daggers and staves. They also are restricted in the armors that they can wear—they can only wear robes and periphery armor (such as boots, gloves, and bracers). In combat, mages must rely on their offensive and defensive spells rather than their fighting capabilities. Mages often have more spell points and fewer hit points than other classes.

```
            The Mage Guild
The mage guild is the guild of magic and magicians. Using their
keen intellect, they are able to call up the forces of
magic to do their bidding. They, like the druids, must pay close
attention to their alignments. Their various spells are very
powerful, but, due to their dedication in the study of their magic,
they have not had time to train in fighting.
```

Monks

Monks specialize in unarmed combat. They need neither weapons nor armor, which tends to makes them pretty self-sufficient. RealmsMUD monks can meditate, which heals them and returns them to full strength. Any interruptions during meditation, however, cancel the effects and the monks must meditate again. Because monks do not need weapons or armor, they tend to have more gold than other players—they can sell all the weapons or amor they find (and they don't have to spend money buying them).

```
            The Monk Guild
The monks' guild is the guild of discipline. They live in seclusion
for much of their lives, learning their art. They are very
adept at using magical spells, and are very proficient fighters.
However, armor interferes with their capabilities, so they are not
allowed to wear it. Their natural armor class is powerful enough,
however, to protect them from any harm.
```

Priests

Priests are the masters of healing and regenerative magic. They can cure typical poisons and diseases, and have the strongest spells for replenishing hit points. They also have a resurrection or reincarnation spell they can use to raise characters from the dead (without the substantial loss of experience points that other methods might result in). Priests can wear most types of armor and can wield a variety of weapons, but their fighting capabilities generally are below average. Priests tend to be pretty popular, however, because of their capability to heal and raise the dead. They tend to work much better in tandem with other players.

```
            The Priest Guild
The priests are men (or women) of the cloth. They spend most of
their time praying to their God. In return, He grants them many
wonderful powers and spells. Due to their dedication to their
God, however, they have not had time to hone their fighting capabilities,
and as a consequence, are not very good at hand-to-hand combat.
But, their many powerful spells more than make up for this deficiency.
```

Thieves

Thieves tend to be some of the least popular players. Among their skills is the capability to steal from monsters (and sometimes players). This means that the thief can take an object the monster has (usually only those items the monster is not wielding or wearing) without fighting the monster. This helps thieves to acquire a large quantity of gold, but tends to make other players mad when their characters spend a long time killing a creature and don't get the treasure because a thief has already stolen it.

TIP

Before you attack a monster (especially a powerful one that might take you some time to kill), you should look at it and make sure any items you expect it to have are still in its possession. If it is a monster you have never fought before, this will not be particularly helpful because you probably will not know what items it should have. If, on the other hand, you are killing this monster solely to get some item that you need or want, check and make sure it has not been stolen or otherwise taken from the monster before you spend a lot of time killing the monster.

```
           The Thief Guild
The thieves are the rogues of RealmsMUD. They slink around, steal
anything they can get their hands on, hide in the shadows, and
have many other capabilities. They are adept at hand-to-hand combat, mostly
because if they get caught, they must fight their way out. Their
abilities come almost naturally to them, and many are dextrous and
strong.
```

NOTE

The preceding example shows another side of MUDs that you will soon notice. MUDs are run by people for fun—no one gets paid for running them. People contribute to MUDs and develop them as a hobby. Because of this, you often will find typos and misspellings. Feel free to report these to wizards, but don't expect any fast changes. The general attitude is "If you can read it, it's good enough." Thus, people usually spend more time working on substantive things and less time on making sure the spelling and grammar are perfect.

Keep in mind this is not always the case. Many wizards and builders are very diligent about the work they do. Many devote a great deal of time to making sure their work is free of spelling and grammatical errors.

This is just a point of information so that you will not be entirely shocked when room descriptions do not appear as professionally written prose or when you encounter the occasional misspelling or the MUD where people are not quite so diligent about the written language.

Again, I want to stress that classes among MUDs can vary widely—not just the types and names of classes that are available, but also the skills and powers they possess. There are MUDs on which priests are very good fighters and MUDs on which mages can wield swords. The best way to learn about classes is to log on to the MUD as a guest and ask people about the different classes (see Chapter 4 for more information about logging on as a guest). I have found that MUDs do make an effort to balance the classes, so your decision should be based on whether you like to adventure alone or in a group (monks, for example, have an advantage in adventuring alone and priests are very powerful as part of a group), whether you like spells or hand-to-hand fighting, and which class you think is the coolest and best fits your MUD persona.

Money and Gold

Most fantasy-based MUDs use gold as the currency system in the MUD world, although this may vary. Some MUDs also have copper and silver as smaller denominations. Some science fiction MUDs use credits as the base currency, and some MUDs even use dollars.

Players can generate money by killing monsters, which often have gold, and by collecting and selling in stores the items these monsters possess. Stores tend to only pay a maximum value for any item, no matter what it is worth (this maximum is often 1000 pieces of gold). Because stores usually do not pay what the item is worth, players often sell weapons and armor to other players to get a better return.

Just like in the real world, money (generally gold) is very important. Once you accumulate some wealth, you can buy weapons in stores when you log on rather than acquiring them from monsters, which generally means killing weak creatures to get small weapons and mediocre armor, and then killing larger creatures to get slightly better weapons and armor. Most LPMUDs do not let you keep your possessions between logins. DikuMUDs, and some LPMUDs, often let you pay rent to store your items between MUD forays.

As you can see, money is important, even in the MUD world. You sometimes can use money to build houses, build castles, or run stores. Some MUDs have more advanced economic systems than others and allow players to own shops and more.

Miscellaneous Information

Your MUD character also may have a *race*, although not all MUDs have races. This race usually is just a cosmetic difference, but it may have other minor effects. Some MUDs give different start stats for different races; thus, an elf may have a slightly higher starting intelligence, a half-orc may have a higher strength, or a dwarf may have a higher constitution. Humans usually are the base characters and have an average score in every stat. Races may have other differences or special skills. On some MUDs, especially DikuMUDs for example, elves and other demi-human races have infravision, which enables them to see in the dark. (Infravision is the ability to see into the infrared spectrum.)

Races also sometimes affect the classes that are available. On some MUDs, for example, only humans can be druids. On DikuMUDs this is very important; DikuMUDs allow characters with multiple classes (the combination of classes available to a new character is defined by his or her race).

Race also defines another characteristic shown in the example of the `score` command, which is *size*. Size is not a standard MUD characteristic, but on this particular MUD it defines the basic size of the character's body. This is important for wearing armor. For more information about armor and size, refer to Chapter 2.

Finally, alignment is the last important characteristic discussed previously. *Alignment* is your outlook on the world—usually good or evil, but sometimes chaos or law. Often, your alignment is based exclusively on what type of monsters you kill—if you kill good monsters, you will be evil; if you kill evil monsters, you will be good. As you can see, it doesn't always have a foundation in reality—sometimes it is nearly impossible to tell what alignment a monster may be.

On some MUDs you can choose your alignment. The alignment you choose will affect your character in several ways. There are weapons on many MUDs that must be wielded by characters of specific alignments. The Holy Avenger sword, for example, is a powerful magic sword that you are only allowed to wield if you have a good alignment. Your alignment also may restrict your ability to enter certain regions of the MUD. Some classes place a heavy emphasis on alignments while for others it is often irrelevant. Priests, more than other classes, for example, usually are affected by alignment. Priests often have different spells, depending on their alignment—and if they deviate from their alignment, they may lose their spells altogether. An evil priest, for example, may have a `cause wounds` spell that does damage to a specific target, while a good priest would have a `heal wounds` spell that would heal damage done to a specific target. If the good priest became evil, he or she might not be able to use the heal wounds spell until he or she returns his or her alignment to good.

Death and Dying

Your MUD character can die. If his or her hit points fall below zero, you will see the following, which is the most dreaded message in the game:

```
You die.
You have a strange feeling.
You can see your own dead body from above.
```

This is the standard LPMUD death message; however, it will be different on DikuMUDs and it may be altered on any given LPMUD. What happens after you die varies.

So your character has just died. How do you get him back? Is he gone forever? On a DikuMUD, your character will be instantly resurrected and transported to the Temple (the central location on most DikuMUDs). Unfortunately, you will lose all your possessions because they will be on your corpse and your newly resurrected body will have only 1 hit point (of course you can be healed to your normal maximum).

Corpses

When your character (and monsters) die, a corpse will be left behind and all of your possessions will be on it. Whenever there is a corpse, you can use `get all from corpse` to take the gold and items that might be on the dead body. This command primarily is used to get the treasure from the monsters that your character kills. Of course, it can also be used when your resurrected character goes to retrieve his items from the dead corpse. And it can just as easily be used by anyone who stumbles across your dead corpse. Note, however, that it is considered bad etiquette to steal from another player character's dead corpse. The proper thing to do is offer to get his stuff and deliver it to his newly resurrected body (wherever that might be) or just leave it there for him to get himself.

Should your character die on an LPMUD, one of two things will happen. You will turn into a ghost and then be teleported to the church (or the central area of the MUD) or left where you are. As a ghost, you really can't do much other than talk to people and wander around. To return to a more solid body, just pray in the church. After you use the `pray` command, you will be resurrected into a new body, although the new body will only have 1 hit point and you will need to heal.

Death on MUDs carries some stiff penalties. On some MUDs, when you die, you lose one or two points from random stats. You also might lose a large number of the experience points you have accumulated—enough to lower you one level, which usually ends up being about a third of the experience points you had before you died. Some MUDs allow priests to resurrect players so that the dead player can avoid this loss of experience points, or suffer a smaller loss. It's a good idea to try to find a priest to resurrect you before using the more standard revival mechanisms.

You will find that death is a real bummer. In fact, I think everyone I know has wanted to break things or has smashed their fist into their desk after a death. After you've been playing for 10 hours to get a level and then die in a moment of foolishness and lose everything, you'll be pretty upset. The first time you die may well be the moment you realize how attached you are to your character.

Player Killing

WARNING

The concept of player killing can be a little unnerving. The term *player killing* has evolved and is a common term on MUDs, but it really has nothing to do with killing other players. Player killing is when players use their characters to kill another player's character. Remember, this is only a game, and if your character is killed by another player's character, don't take it personally.

Many people who are attracted to player killing are the same people who enjoy other head-to-head games like Mortal Combat and Street Fighter. In these games, and on some MUDs, the objective is to kill your opponent. Just remember, it is just a game.

Player killing? I know it sounds odd, but you are in a vast, new world with many other characters. Sometimes violence erupts. It is not uncommon for players to get mad at each other, so what recourse is there? Well, you can complain about it to one of the MUD's administrators (usually a wizard), but if it isn't a problem that violates MUD rules, it is likely that the administration won't intervene. So what do you do?

Just as you can kill monsters, you also can kill players. Killing other players is not something to be taken lightly, however. In fact, on many MUDs it is banned—and on some MUDs, not only is it banned, but the capability to kill other players has been completely removed from the game. On many MUDs, however, it is still legal, although often frowned upon. The attitudes on player killing vary widely, from it should not be allowed to it's the whole game.

On most MUDs, where it is allowed, player killing happens only rarely. This brings up the distinction between player killing and being a player killer. If you kill someone once and have a pretty good reason, it generally won't cause problems (unless that person has powerful friends or wants revenge). If you kill several different players for poor reasons, however, you could start to earn the angst of other players. If you wander around wantonly killing other players, you certainly will attract the attention of a MUD lynch mob.

Some players enjoy player killing because it is very different from killing monsters. Because MUD monsters usually aren't very intelligent, monsters don't strategize and usually don't chase after you if you run away. Fighting monsters is more of knowing when it is dangerous for you to continue the fight so that you do not die. Because other players are unpredictable, some players enjoy the challenge and the greater risk involved in player killing.

The allure of player killing is so great that several MUDs have developed the focus solely on player killing. They have no normal avenues for adventuring and all players are equal in power. The only objective is to kill everyone else before you die.

Genocide

This LPMUD is devoted to player killing. One of the first, and certainly most popular, player killing MUDs, Genocide is always packed full of players. Genocide has a system of wars. If you log in when a war is in progress, you will be dead and ineffective until the next war. At the end of the war, everyone is brought back to life and given the same stats, hit points, spell points, and so on; thus, everyone is equal and a new war starts and all players run off into the MUD world to collect as much money as possible and find the best weapons and armor (to more effectively kill other players). It's a race to find the best items and the most money. Knowing the geography of the MUD is very important because this knowledge (knowing where to run) can save you, and knowing where to find the best equipment can help you.

There also are team wars in which the players online are divided into even teams (based on wins and kills in previous wars) and then are released to go kill the members of the opposing team(s).

This system of wars can be a lot of fun. It is very fast paced and is a great diversion. If you are killed on another MUD, it's nice to go to Genocide and vent your frustrations. And dying on Genocide doesn't hurt you—unless, of course, you are a die-hard Genocide junkie, which there are many, and your kill-to-death ratio is very important to you.

Address	`genocide.shsu.edu or camelot.shsu.edu (192.92.115.145)`
Port	2222
Type	LPMUD (MUDs)

Summary

Now you understand the basics of MUD characters and what they're about. This should give you the basics you need to jump on a MUD and see what it is really like. Onward to Chapter 4 and connecting to MUDs!

4 CHAPTER

Connecting To MUDs

Now that you understand the basics of MUDs, it is time to connect to one. Before connecting to a MUD, however, you have to know a little about the Internet. The *Internet* is a huge web of interconnected networks that spans the globe. Originally made up of government and education sites, it now encompasses huge numbers of commercial sites, and the number of users that access it is increasing astronomically. The Internet relies on many protocols (formal structure for exchanging information between two or more computers) to provide users with access to its many resources. One protocol, called *TCP/IP* (Transmission Control Protocol/Internet Protocol), provides a roadway for all of the other protocols to communicate. TCP/IP works much like the highway system: TCP/IP and the physical connections of the network are the road, the other protocols are cars and trucks, and the information you are sending and receiving is riding in the cars and trucks. Just like cars, there are many different protocols, such as *HTTP* (HyperText Transfer Protocol), the protocol used by the World Wide Web and Mosaic; the Gopher protocol; and the FTP (File Transfer Protocol), which is the protocol used for transferring files.

MUDs rely on another protocol called *telnet*, which allows the user to connect to other machines on the Internet. Most of these computers on the Internet are not Macintoshes or PCs but multi-user machines running operating systems such as UNIX. Because these machines support large numbers of users, they require that you log in. The login process basically means that you have to identify yourself to the computer so that it can grant you the correct privileges and access. Your identity is your *username* (often your initials or your last name) and a password. Because only you know your password, your identity is confirmed and prohibits others from mimicking you on the Internet. This safeguard is very important on MUDs because you don't want someone walking around the MUD pretending to be you.

Depending on what kind of connection you have to the Internet, telnet may have several different guises. If you connect to a UNIX machine via a modem (generally called a dial-up or shell account), you likely know how to use telnet already. However, if you don't know how to use telnet, all you need is a simple command. From the prompt (probably a `%`, a `$`,. or something ending in `>`), type **`telnet helios.cs.duke.edu`**. You see `Trying 152.3.145.1...`, which indicates that the computer you are using is trying to connect to the computer to which you have just tried to telnet.

The `telnet` command comes in two main varieties:

`telnet` *`<machine name or number>`* *`<port>`* for UNIX-based machines

`telnet` *`<machine name or number>`* `/port=`*`<port>`* for VAX/VMS-based machines

`<machine name or number>` is the computer to which you want connect (telnet), such as `helios.cs.duke.edu` or `152.3.145.1`

`<port>` is the port number (which defaults to 23). MUDs use non-standard port numbers, so you will need to type in a separate number. In `telnet realms.dorsai.org 1501`, for example, `1501` is the port number of the MUD

The following is an example of a standard telnet session:

```
%telnet helios.cs.duke.edu
Trying 152.3.145.1...
Connected to helios.cs.duke.edu.
Escape character is '^]'.


SunOS UNIX (helios)

login: busey
Password: [You will not see your password as you type it.]
```

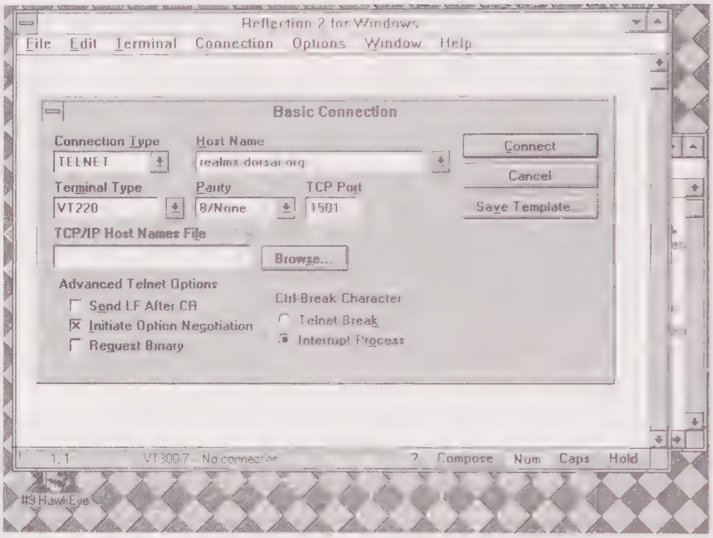
If you are using *SLIP* (Serial Line Internet Protocol) or *PPP* (Point-to-Point Protocol) to connect to the Internet (SLIP and PPP enable your computer to simulate a direct connection to the Internet over a modem), or if you are lucky enough to have a direct connection, you have to launch an application to telnet. This application is likely called telnet and is included in every commercial TCP/IP package, such as NetLink's Internet Access Kit, Spry's Internet in a Box, and FTP Software's OnNet. The telnet application will most likely have an open connection option that allows you to type in the address of the MUD or the computer with which you want to connect. Using these custom telnet applications is usually straightforward. Because the applications can vary, consult the manual if you have any problems.

Ports

MUDs work in a slightly different way but within the parameters of telnet. Because MUDs run on computers that are usually being used for other purposes (such as supporting a school's e-mail server, a company's World Wide Web server, general login for working under UNIX, and so on), they must be reachable separately from the computer itself. For example, there may be a MUD on `realms.dorsai.org`.... By typing **`telnet realms.dorsai.org`**, you get the login prompt for the computer's operating system (usually UNIX), which is different than the login prompt for the MUD. MUD addresses generally have a second component called the *port*. The port is added at the end of the telnet address to identify the MUD. For example, `telnet realms.dorsai.org 1501` reaches the MUD instead of the computer's operating system. The port also becomes relevant when a computer is used to run more than one MUD. For example, `telnet marble.bu.edu 5000` takes you to an LPMUD called 3 Kingdoms, while `telnet marble.bu.edu 7777` takes you to a CircleMUD called HexOnyx.

If you are using a telnet application (as with TCP/IP, SLIP, or PPP), you may have a separate port entry when you open a connection. This port entry probably contains the number 23 as a default (or the word *telnet*). The number 23 is the standard port number for connection directly to the machine's operating system; 23 is also the assumed port when you telnet without a port number. So if you are using your telnet application and a `23` shows up in a box or window for the port, just type the MUD's port number over the 23.

Figures 4.1 and 4.2 show screen shots from two telnet programs for Microsoft Windows. As you can see, both figures have different screens for setting up the parameters for a telnet connection. It is fairly easy, however, to discern where to enter the port number for each program.

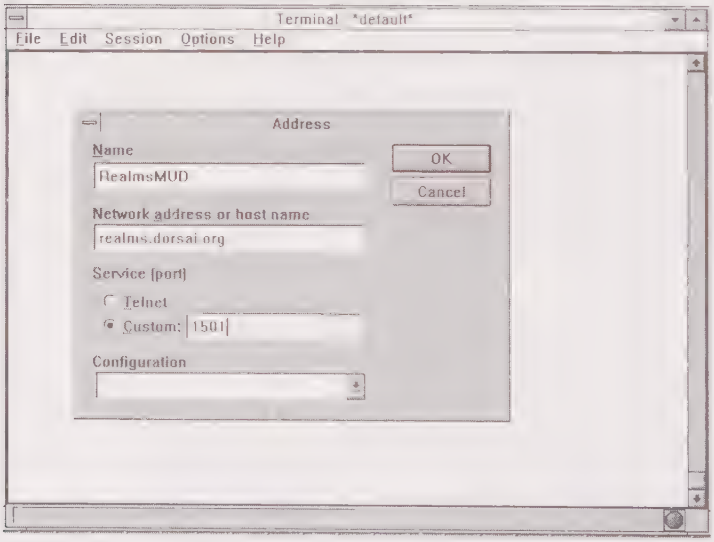

Basic Connection

Connection Type: TELNET
Host Name: realms.dorsai.org
Terminal Type: VT220
Parity: 8/None
TCP Port: 1501
TCP/IP Host Names File:

Advanced Telnet Options
- [ ] Send LF After CR
- [x] Initiate Option Negotiation
- [ ] Request Binary

Ctrl-Break Character
- ( ) Telnet Break
- (●) Interrupt Process

Figure 4.1.
The connection screen for Reflection 2, a terminal emulation program (which includes telnet) from Walker Richer & Quinn. This is a commercial program and allows many different types of network connections and terminal emulations.


Address

Name: RealmsMUD
Network address or host name: realms.dorsai.org
Service (port)
- ( ) Telnet
- (●) Custom: 1501

Configuration:

Figure 4.2.
EWAN, a good, freely distributed Microsoft Windows telnet program. Note, however, that companies and others needing support are asked to pay $495 a year to the author.

Both these programs work with any Winsock 1.1-compliant TCP/IP stack, which includes the Trumpet Winsock stack (which is available on the Internet) and any of the other commercially available products, such as NetLink's Internet Access Kit and Spry's Internet in a Box. A stack is a special piece of software that provides your computer with TCP/IP and other Internet protocols.

NOTE

You can get the latest version of the EWAN telnet program (shown previously) for Microsoft Windows by using a WWW browser to connect to the following:

`http://www.lysator.liu.se/~zander/ewan.html`

You also can use anonymous FTP to connect to the following:

`ftp.lysator.liu.se`

You can get this file (in binary mode) by typing the following:

`/pub/msdos/windows/ewan1052.zip`

The listed file is for EWAN, Version 1.052, which is current as of the writing of this book.

I recommend using the World Wide Web. It has pointers to sites in the U.S. and Canada where the latest versions of the software are available. The FTP server is in Sweden and could potentially be very slow for U.S. users.

What You Will See

Now that you know how to connect to computers on the Internet, and more importantly to MUDs, take a look at what you actually see when you log in to a MUD.

```
%telnet realms.dorsai.org 1501
Trying 198.3.127.200...
Connected to realms.dorsai.org.
Escape character is '^]'.

                                           _   __,----'~~~~~~~~~~'----.__
                                     .  .    '//====-_            ___,-'  '
                    -.            \_|// .   /|||\\ '~~~~'--.___./
              _______.==.      _-~O~  \/    |||  \\           _,''
        __,--'   ,=='||\=_    ;_--~/_-'|-   |'\   \\        ,'
     _-'      ,='     |  \\'.    '-'~7 /-   /   ||  '\.     /
   .'       ,'        |  \\ \_     / /-   /    ||      \  /
  / ______ /          |    \\.'-_/  /|- _/   ,||        \ /
 ,-'     '-|--'~~'--_ \      '==-/  '| \'--===-'      _/'
            |         `-|      /|   )-'\~'      _,--~'
                        '-~~\_/ |   |   `\_    ,~             /\
                             /  \     \___ \/~                '\___
                        _,-' _/'\  ,-'~_____-''-/                "===\
                        ((->/'    \|||' '.    ~'-/  ,               _||
+~~~~~~~~~~~~~~~~~~~~~~~~~~~+      \_    ~\      '^--|__i__i__\--~'_/
|     The Return of         |    __-^-_     ')  \-.________________,-~'
|       RealmsMud           |    //,-'~~'__--^- |-----------'
|                           |          //,--~~'-\
'~~~~~~~~~~~~~~~~~~~~~~~~~~~'     (Art via USENET)

     Use the name: 'guest', password: 'realms', if you just want a look.
```

```
DISCLAIMER:  The administrators of this MUD reserve the right to monitor and/or
log any activity that takes place on it. They further reserve the right to
restrict access to anyone for any reason, without notice.

Amylaar Version: 03.02@310

Name & then password to enter the game. Enter or 0 to leave.
1 then name and password so you can change your password.
2 to see who is currently playing.

What is your command? tarod
Password: [You will not see your password when you type it.]
You get the player object.
Connecting to obj/player ...
```

If you are using the standard command line telnet, you can do several things to make your MUD experience more interesting. All of these things can be mimicked in telnet applications as well, but the commands for using them vary widely.

Suspending Your Session

The command `Control-]` returns you to a `telnet>` prompt that provides many things. The most useful command is `z`. By using `z` at the `telnet>` prompt, you return to your UNIX command prompt, where you then can `finger` people. (`finger` is an Internet/UNIX command that enables you to find information about other users.) You also can send mail or open another MUD session. After you do whatever you need to do in UNIX, you can type **`%1`** (or on some machines **`fg 1`** or **`fg %1`**) to return to the open MUD session. This procedure works numerically; for example, if you have two MUD sessions open, you use **`%1`** and **`%2`** to go to those sessions. (`%1` is the first MUD you have connected to, and `%2` is the second, and so on.)

The following is an example of how this might look. This sample session starts while I was logged in to a MUD. I used a MUD command (`smile`) to show that I was on the MUD. I then used `control-]` (which is shown as `^]`) to return to the UNIX prompt. I then used the UNIX command, `finger`, to get information about myself. When I was done, I used `%1` to return to the MUD.

```
> smile
You smile happily.
> ^]
telnet> z

[1]+  Suspended                telnet realms.dorsai.org 1501
matrix:/usr/home/busey>finger busey
Login: busey                             Name: andrew busey
Directory: /home/busey                   Shell: /bin/bash
```

```
On since Mon Apr 17 12:52 (CDT) on ttyp1 from 199.171.21.122
No Plan.
matrix:/usr/home/busey>%1
telnet realms.dorsai.org 1501
> wink
You wink.
>
```

Logging in to Your First MUD

Now that you have a sense of how MUDs fit into the Internet as a whole, look at logging in to an actual MUD. Logging in to a MUD for the first time can be an adventure. Most MUDs enable you to login as `guest`. Guest accounts usually do not require passwords, but try `guest`, if you do need one.

People have a wide variety of tastes in MUDs—just like anything else—and it often takes looking at a few MUDs before you find one you like and are comfortable playing. Using the `guest` character is a good way to narrow the search. If, after you login as `guest`, you decide that it is not the MUD for you, you can just move on to the next one. Using the `guest` character to look around also is a courtesy to the MUD administrators who do not want tons of new characters who have only logged in once filling up their system.

To help you understand the different things you may encounter, the following sections walk you through the process of logging in to three different types of MUDs.

Passwords

When you first log in to a new MUD to create your MUD character, you will need to choose a password. Your password is very important. Your password is the only thing that stops someone else from logging in to the MUD and assuming your character. Imagine all the evil someone could do if they could assume your real life body for a day. That's what it would be like in the MUD world if someone got your password!

Choosing a password for a MUD is similar to choosing a password for any computer account. Choose a word, or better yet, a phrase or anagram, that is not obvious. Do not, for example, use the name of your character, your first name, or the name of your significant other. And never, never use the same password as the one you use on other MUDs and any other Internet computer accounts you may have. Most MUDs prevent players from getting the passwords from within the MUD, and most encrypt the password when it's stored in the database files. However, there is nothing preventing the MUD's owner from modifying the code to dump the passwords to a file, along with other information such as the host from which you connected. Using this information, an evil MUD admin probably could figure out your login name and easily get into your account.

You also should not use the same password on different MUDs. If your password gets out on one MUD, all your MUD characters will be compromised. This especially is important for MUD Wizards and Gods. If your client has a auto-login feature, you should use it to protect the file that contains the login information from being read by others.

Possible Password Problems

This story talks about a program called Crack. *Crack* is designed to take dictionaries (whether of the Webster's variety or just a collection of words, phrases, or existing passwords) and check them against the list of usernames and passwords on a UNIX computer. Crack can find any passwords on that UNIX computer that are in the dictionaries that Crack has been configured to use.

The following story comes from Alec Muffett, author of Crack and maintainer of the `alt.security FAQ`, and is taken directly from the MUD FAQ that is compiled by Jennifer Smith.

> `aem@aberystwyth.ac.uk`: The best story I have is of a student friend of mine (whom I will call Bob) who spent his industrial year at a major computer manufacturing company. In his holidays, Bob would come back to college and play AberMUD on my system.
>
> Part of Bob's job at the company involved systems management. The company for which he worked was very hot on security, so all the passwords were random strings of letters and had no sensible order. It was imperative that the passwords were secure (this involved writing down the random passwords and locking them in large heavy-duty safes).
>
> One day, on a whim, I fed the MUD persona file passwords (which were stored in plain text) as a dictionary (one of the custom ones mentioned in the introductory paragraph above) into Crack and then ran Crack on our system's password file. A few student accounts came up, but nothing special appeared. I told the students concerned to change their passwords—that was the end of it. (That was the end of Crack's testing on that computer, but the dictionary containing the MUD passwords was never removed, which is explained in the next paragraph.)
>
> Being the lazy guy that I am, I forgot to remove the passwords from the Crack dictionary, and when I posted the next version to Usenet, the words also were posted. It (Crack) went to the `comp.sources.misc moderator`, came back over Usenet, and eventually wound up at Bob's company. Round trip: 10,000 miles.
>
> Being a cool kinda student sysadmin dude, Bob ran the new version of Crack when it arrived. When it immediately churned out the root password on his machine, he nearly fainted. The moral of this story is never use the same password in two different places—especially on untrusted systems (like MUDs).

Logging in to Your First LPMUD

LP MUD

RealmsMUD is an *LPMUD*. Like most LPMUDs, RealmsMUD has a significant amount of custom features that have been added over the years of its existence (over three years). LPMUDs vary significantly, but after you get the basics down, you can easily navigate the different incarnations.

```
%telnet realms.dorsai.org 1501
Trying 198.3.127.200...
Connected to realms.dorsai.org.
Escape character is '^]'.


                            /(_  /(_
                           /   \/   \
             ¦\___/¦      //¦¦\//¦¦ \\       Realmsmud, The Original
(,\  /,)\__  // ¦¦// ¦¦ \\ \                (USENET art)
            /     /   /_//  ¦//  ¦¦  \\ \\
           (@_^_@)/    /_   //   ¦¦   \\  \\
            W//W_/      /_ //    ¦¦    \\   \\
          (//) ¦         ///     ¦¦     \\    \\
        (/ /) _¦_ /   )  //      ¦¦      \\   __\
      (// /) '/,_ _ _/  ( ; -.   ¦¦    _ _\\.-~        .-~~~^-.
    (( // )) ,-{        _      '-¦¦.-~-.           .~         '.
   (( /// ))  '/\      /                 ~-. _ .-~      .-~^-.  \
   (( ///))      '.   {            }                   /      \  \
    ((/ ))     .----~-.\        \-'                 .~         \  '. \^-.
              ///.-----..>    (  \             _ -~             '.  ^-'   ^-_
                ///-._ _ _ _ _ _ _}^ - - - - - ~                  ~--_.   .-~
                                                                      /.-~

DISCLAIMER: The administrators of this MUD reserve the right to monitor and/or
log any activity that takes place on it. They further reserve the right to
restrict access to anyone for any reason, without notice.

Amylaar Version: 03.02@310

Name & then password to enter the game.
Enter or 0 to leave.
1 then name and password so you can change your password.
2 to see who is currently playing.

What is your command? justicar
New character.
```

NOTE

To create a new character on an LPMUD, just type the name of your new character at the first prompt. The prompt might be `What is your command?`, `login:`, or `Enter your name:`, depending on the MUD administrators.

```
Password: [Type your new password here; you usually will not see it when you
type it.]
Password: (again) [Type your password again to make sure there were no mis-
takes.]

***** Realms Mud *****

0) Exit the game.
1) Enter the game.
2) Change password.

  Enter thy choice: 1
You get the player object.
Connecting to obj/player ...
Welcome to Realms, the BEST LPmud EVER!!!
You get 1 points to spend on your stats.
You are now Justicar the utter novice (level 1).
Please enter your email address (or 'none'): none
Pick a race, (?) for a list:
?

Valid Races
----------
Human
Elf
Dwarf
Halfling
Gnome
Half_orc
Faerie
Half_elf

Pick a race, (?) for a list:
elf
Are you, male or female: male
Welcome, Sir!

                          do "help NEWS"
       (You are responsible for it, even if you have not read it)
***********************************************************************

We updated the machine and parser! Please if you find bugs that could be
parser/machine related ONLY, _mud_ mail Animal.

*MANY* thanks to Dorsai for the net connection!
*MANY* thanks to all the people involved who made it possible!!!

REALMS Class Information (read sign).
> l
This is the initiation room.
     Welcome to RealmsMud. This is the start room.
You may now spend points on your stats. You do this by
typing: spend <stat>. ie spend int  Once this is done
```

```
it is recommended that you wander around and check out
the guilds before you choose to 'join' a guild.

Good Luck and Enjoy!

The commands in this room are: points, spend, done.
REALMS Class Information (read sign).
```

The sign mentioned in the preceding sample session is the same sign mentioned in Chapter 3 in the segment titled "The Class or Guild." On RealmsMUD, you do not choose your class in the login sequence; you choose it after you begin playing.

```
> points
Points Left: 3
```

The preceding text indicates that I have three points left. On this particular MUD, you are given three points to add to your character's starting stats. Your starting stats on this MUD are two plus whatever modifications your race may give you. Other MUDs use slightly different systems for creating your character's initial stats. Other systems include complete random starting stats and different base stats.

```
> spend con
You bought a point of constitution.
```

TIP

`con` is short for constitution and `int` is short for intelligence. Constitution increases your hit points and intelligence increases your spell points. I have found that on the majority of MUDs, these by far are the most beneficial stats. You should increase your constitution, and if you plan to use magic, you should increase your intelligence.

```
Points remaining: 2
> spend int
You bought a point of intelligence.
Points remaining: 1
> spend con
You bought a point of constitution.
Points remaining: 0
> score
 Justicar the utter novice
-------------------------------------------------
```

```
Level: 1
You have 0% of the experience needed for the next level.
Hp: 56/56       Sp:1/56
Money: 0    (11000)
Guild: none
Race: elf
Size: medium
Alignment: neutral
Str: 2 Int: 5 Wis: 2
Dex: 4 Con: 4 Chr: 4
Wimpy 0%
Hunted by: Nothing at all, aren't you lucky?
You are normal. (Yeah, right)
> done
The Church

You are in the main church of Realmsmud.
You see a set of stairs that go down to the healing waters of
the Realms. There is a huge pit in the center, and a door in
the west wall. There is a button beside the door.
There is a clock on the wall.
This church has the service of reviving ghosts. Dead
people come to the church and pray.

******DON'T CAST SPELLS OR FIGHT IN THE CHURCH!*****

There are exits south, north, up, east, and down.
Zxaigon the utter novice (Mortal).
A magic portal, leading to many houses.
REALMS players rules.
```

Virtually all LPMUDs start in the church. The church is important because it serves as a central location within the MUD. The church also is a haven, which means that no combat is allowed while you are there. Finally, if you die, you can come to the church and pray. After you pray, you are resurrected, although not without a penalty in experience points and stats. Before you pray, however, you might want to look for a priest who can resurrect you without the cost in experience points that the church will exact.

```
> read rules
Welcome to Realmsmud!

Hi, and welcome. This is a short but sweet list of rules. These
are enforced, so it probably wouldn't be a wise idea to ignore them.

1) No cheating on quests.
This includes getting help from other people while doing a quest. The
whole
concept of quests is to test your problem-solving abilities. And
that doesn't work if someone tells you how to do the quest from
beginning to end. (plus, it's not nearly as fun.. ;>  )
```

```
2) No shouting profanities.
This is kind of objective, but I trust you all know what bad words are.. ;>
Shouting also includes guild and race lines, so have fun, but watch
your language while doing it. (keep it halfway civil)

3) No Player Killing (except by those registered)
Unless you are registered, or the owner of a diamond, do _NOT_
kill other players, by any means, understood??  And for diamonds
if you go around reapering lower level players, or use it too much
for no apparent good_reason, then you will be reapered (more than a
few reapers at once), or electrocuted, or both.

4) Be Nice
This is pretty much common sense. It makes the mud a whole lot
more fun for everyone if you're nice. Just treat everyone the way
you would in real life.

5) Cheating/Bugs
Cheating of any kind will not be permitted. Right now this mud
is going through a transition period. There WILL be bugs, taking
advantage of them and not reporting them is a nasty offense and
WILL be enforced. So if you see a bug, let us know by typing
bug <then the problem>. (or telling/mailing the wizard whose bug
it is)

6) Harassment
Harassment of any kind on this mud will also _NOT_ be tolerated. If
you can't get along with someone here, then don't play.

7) Robot
NO ROBOTS or ROBOT type actions will be tolerated. Minor
client interaction is permissible, like picking up stuff if
someone drops something. But NO auto-killing modes, no running
through the mud full speed till your client spots a live monster
and stops for you. Anyone caught with this (and it's REAL easy
to catch you) will be cut in half in everything, including
your level. Going from level 30 to 15 sure would suck....

7) HAVE FUN!!!
BY FAR, the MOST important rule.
> s
You are in an open area south of the village church. To the east
is a substantial town. Forest blankets the hills to the west.
You can see the top of a massive board through the trees to the south.
There are three obvious exits: west, east and south.
Grudge i am finally back to positive exp. #=-) (Mortal).
Minstrel Rabidchild died and lost tons of xp, but is helping newbies
anyway!  Go figure! (Mortal).
Xtreme the utter novice (Mortal).
A Short Dream Post.
Grudge leaves east.
> e
A track going into the village. The track opens up to a road
to the east and ends with green lawn to the west. You notice
a small hole here.
```

```
There are three obvious exits: north, west and east.
Taishan is getting engaged sooner than you may think....:) (Mortal).
A small hole leading down—Newbie Area.
```

Watch for `A small hole leading down—Newbie Area.` (newbie areas, not small holes); they can be very helpful to a new player. A newbie is a name for new players, like frosh or freshman is a name for new college students. Newbie areas usually contain monsters that you have a pretty good chance of killing without being mauled. Because these areas do not allow higher-level players, the newbie areas are less likely to be completely plundered.

Logging in to Your First DikuMUD

DIKU MUD

Elite is a popular MUD based in Sweden. It is based on DikuMUD and one of its derivatives, CircleMUD. Although it is customized, it is used here as the reference for DikuMUDs. Just like with LPMUDs, DikuMUDs vary significantly. After you get down the basics, however, you can easily navigate the different incarnations.

```
========================================
%telnet 130.237.222.237 4000
Trying 130.237.222.237...
Connected to xbyse.nada.kth.se.
Escape character is '^]'.

                      ____/    /       / __   __/    ____/
                     /        /       /      /      /
                    ___/     /       /      /      ___/
                   /        /       /      /      /
                _______/ ______/ __/    __/    _______/

                         - MUD 5.1 with AQS

           Under Creation by Petrus Wang & Richard Rosenberg.


                      Based on DikuMUD (GAMMA 0.0)
                               Created by
                  Hans Henrik Staerfeldt, Katja Nyboe,
           Tom Madsen, Michael Seifert, and Sebastian Hammer

         Special credits to Jeremy Elson for CircleMud2.2 code

By what name do you wish to be known? tarod
Did I get that right, Tarod (Y/N)? y

New character.
Give me a password for Tarod: [You will not see your new password as you type
it.]
```

```
Please retype password:  [You must type your new password again to make sure
you there were not any mistakes.]

Select Screen Mode:
  [a] Normal Terminal
  [b] VT100 Compatible
  [c] VT & ANSI Color
  [d] IBM/PC color&character set
  [?] Help!
Screen Mode: a
```

The ANSI color and VT100 emulation on this MUD are very nice. Unfortunately, unless you have a very high-quality terminal program (if you are using a shell account to access the MUD) or telnet application (if you have a direct or SLIP/PPP connection), you may find that it does not work very well. When it does work, it is very convenient and adds a lot to the MUD; but when it goes on the fritz, it can be a big problem. (Make sure the ANSI and VT emulation you have are solid, which you will find to be true in most commercial software.) If you don't know about terminal emulations, you probably shouldn't try this.

```
What is your sex (M/F)? m


Select a race:
  [a] Human       [b] Half-troll  [c] Halfling
  [d] Dwarf       [e] Gnome       [f] Elf
  [g] Half-elf    [h] Half-ogre   [i] Half-orc
  [j] Duck        [k] Fairy       [l] Minotaur
  [m] Ratman      [n] Drow        [o] Lizardman

Enter capital letter to get info about race
Race: f
```

The races available in this MUD are quite diverse. The races on DikuMUDs also often tend to mean more than on LPMUDs. For example, elves can have *infravision* (the capability to see in the infrared spectrum) on DikuMUDs and, therefore, can see in certain dark areas where others cannot (without some external light source.) The specific capabilities may vary, but expect some racial capabilities on DikuMUDs, whereas on LPMUDs, the race will only modify your stats and your social situation.

```
Select Class:
  [a] Magic-user
  [b] Cleric
  [d] Warrior
  [g] Bard
  [h] Knight
  [i] Wizard
  [j] Druid
  [l] Ranger
  [n] Paladin
  [q] Warrior/Thief
  [r] Warrior/Cleric
  [s] Warrior/Magic-user
  [t] Thief/Cleric
  [u] Thief/Magic-user
  [v] Cleric/Magic-user
  [-] Reselect Race

Enter capital letter to get info about class
Class: s
```

As you can see in the preceding sample session, this DikuMUD has are a large selection of classes. As is common on DikuMUDs, there also is the option to be multi-classed. Also, DikuMUDs restrict certain classes and multi-class combinations to certain races. In the preceding, if the user chooses to be human, the list of classes presented is different.

```
Project EliteMud started 9th April 1994.

Elite is a serious programming project, with the intention of
research and testing of algorithms, datastructure and complexity
with multi-user systems.

Elite is under development. That means there may be some bugs.
The balance of the mud may not be too good either, though hard
work has been put in on that.
As a player, you will have things to say about this mud too.
All ideas are welcome.

*  This is NOT a renting mud. Equipment saved on quit.
*  The stats are no longer rolled:  All start with 11 and with 13 in
two prime stats. Stats will improve (not always) when leveling.

*  CLAN SYSTEM IN—Use CLAN command
*  All changes/news will now be put in NEWS.
Read the NEWS often with : > NEWS  (CAPITAL LETTERS)

" It's not that I am afraid to die, I just don't want to be there
when it happens. "
- Woody Allen


*** PRESS RETURN:
```

TIP

After creating a new character, you should use Option 2 to enter a description. Adding a description gives your character more depth.

```
                   (  _) /  o _/_ _    /¦/¦       ___)
                   _)   /  /  /  /_)  / ( ¦  / / /  /
                  (____(__(__(__(____/    ¦_(_(_(__(

                  (0) Quit EliteMud.
                  (1) Adventure in the Realm of EliteMud.
                  (2) Enter description.
                  (3) Read background story.
                  (4) Change password.
                  (5) Delete this character.

                  The option is yours: 2

Enter the text you'd like others to see when they look at you.
Terminate with a '@' or '.' at the beginning.
Old description :
] Tarod is a tall elf with long black hair and glowing green
] eyes that seem to focus a hidden inner power.
] .
```

After entering a description for your new character, you once again are returned to the login screen, but now you can enter the MUD and begin your adventure.

```
                   (  _) /  o _/_ _    /¦/¦       ___)
                   _)   /  /  /  /_)  / ( ¦  / / /  /
                  (____(__(__(__(____/    ¦_(_(_(__(

                  (0) Quit EliteMud.
                  (1) Adventure in the Realm of EliteMud.
                  (2) Enter description.
                  (3) Read background story.
                  (4) Change password.
                  (5) Delete this character.

                  The option is yours: 1

Welcome to the land of EliteMUD!  Stay awhile ... Stay FOREVER!

The Temple Of Midgaard
   You are in the southern end of the temple hall in the Temple of
Midgaard.
The temple has been constructed from giant marble blocks, eternal in
appearance, and most of the walls are covered by ancient wallpaintings
picturing Gods, Giants and peasants. Large steps lead down through the grand
temple gate, descending the huge mound upon which the temple is built and ends
on the temple square below. To the west, you see the Reading Room. The donation
room is in a small alcove to your east.
A large, sociable bulletin board is mounted on a wall here.
An automatic teller machine has been installed in the wall here.
Firebird The Vulcan's Fire Sword is standing here.
Mistress Nymph will do anything if you WORSHIP her! is standing here.
```

```
Trentreznor the Lizard Magic-apprentice is standing here.
Funnyguy the Minotaur Master-Blade/Patriarch is sleeping here (Zzzzzzz).
Lyioness the Half-elven Novice/Spell-delvress is resting here.
Gelon the Drow Swordpupil/Believer/Magic-apprentice is standing here.
Raster the Half-elven Knight/Patriarch/Minor Elemental is sleeping here
(Zzzzzzz).
Toyota the Minotaur Rogue/Minister (linkless) is sleeping here (Zzzzzzz).
+A horse is standing here.
+The Priest is standing here, offering his services.
[Exits:neswd]
You don't have an alias file.

< 35Hp 110Mn 93Mv >
Firebird walks east.

< 35Hp 110Mn 93Mv >
Nelric walks in from the east.

< 35Hp 110Mn 93Mv > stats
Level 1—Tarod the Elven Swordpupil/Magic-apprentice -
17 year old male elf 2-multi-class player
Levels 1/1 warrior/magic-user
Str: [13/0]  Int: [11]  Wis: [11]  Dex: [11]  Con: [13]  Cha: [11]
AC[100/10] Hitroll[ 0] Damroll[ 0] THAC0[20] Resistances[0/0/0/0/0]
Magic: (innate)    infravision
Wolfe walks south.

< 35Hp 110Mn 93Mv > score
You are Tarod the Elven Swordpupil/Magic-apprentice (level 1).
You are a 17 year old elf. It's your birthday today.
You have 35(35) hp, 110(110) mana and 93(93) movement points.
You are neutral.
You have scored 1 exp, and have 2000 gold coins.
You need 2665 exp to reach your next level.
You have been playing for 0 days and 0 hours.
You are not carrying anything.
You have nothing in your inventory and no items equipped.
You are standing.

< 35Hp 110Mn 93Mv >
Sixxgunn appears in the middle of the room.

< 35Hp 110Mn 93Mv > skills
You have got 3 practice sessions left.

These are the skills you know:
stab                  (not learned) ¦ bludgeon                (not learned)
slash                 (not learned) ¦ chop                    (not learned)
spellcraft            (not learned) ¦ pierce                  (not learned)


< 35Hp 110Mn 93Mv >
```

Logging in to Your First MOO

LambdaMOO is the most popular MOO (perhaps the most popular individual MUD of any type), and it has received a lot of publicity (it has even been covered in *Newsweek* in the November 7, 1994 issue). Because of this publicity, it also is very crowded (often over 150 users) and can be slow. For example, the following session has 13 seconds of lag time. MOOs work a little differently than the two types of MUDs (LPMUDs and DikuMUDs) discussed earlier. To become an official MOO player, you need to request that a character be made for you (whereas on the MUDs that you have seen in the last two example, your character was created instantly). To create your character, for example, you may need to send e-mail to an administrator or use an application process that is part of the MOO (which is the case on LambdaMOO).

```
%telnet lambda.xerox.com 8888
Trying 192.216.54.2...
Connected to lambda.xerox.com.
Escape character is '^]'.
                           ****************************
                           *  Welcome to LambdaMOO!   *
                           ****************************

                       Running Version 1.7.8p4 of LambdaMOO

PLEASE NOTE:
   LambdaMOO is a new kind of society, where thousands of people voluntarily
come together from all over the world. What these people say or do may not
always be to your liking; as when visiting any international city, it is wise
to be careful who you associate with and what you say.
   The operators of LambdaMOO have provided the materials for the buildings of
this community, but are not responsible for what is said or done in them. In
particular, you must assume responsibility if you permit minors or others to
access LambdaMOO through your facilities. The statements and viewpoints
expressed here are not necessarily those of the wizards, Pavel Curtis, or the
Xerox Corporation and those parties disclaim any responsibility for them.

For assistance either now or later, type 'help'.
The lag is approximately 13 seconds; there are 171 connected.

help

Type 'connect <character-name> <password>'     to connect to your character,
'connect Guest'     to connect to a guest character,
     'create'            to see how to get a character of your own,
     '@who'              just to see who's logged in right now,
     '@uptime'           to see how long the server has been running,
     '@version'          to see what version of the server we're running, or
     '@quit'             to disconnect, either now or later.

For example, 'connect Munchkin frebblebit' would connect to the character
'Munchkin' if 'frebblebit' were the right password.

After you've connected, type
     'help'              for more documentation.

Please email bug/crash reports (but NOT character-creation requests)
```

Typing **help** displays list commands available from the login screen. The results of create and connect guest are shown in the following paragraphs. To connect by using an actual player, you need to log in as guest and send a message. @who is not very useful on LambdaMOO because it does not show you who is 1 because usually there are over 100 people on the MOO at any given time. @quit disconnects you from the MOO. @uptime and @version are one line pieces of information that do not serve much of a purpose.

```
create
```

To get a character, log in as a guest user and use the command @request *<character-name>* for *<email-address>*. The character is entered in the waitlist. The password is mailed to the e-mail address when the character is created. After you are on as a guest, read *b:mpg for details of the waitlist mechanism. Note that only one character per person is allowed.

```
connect guest

Okay,... guest is in use. Logging you in as 'Plaid_Guest'
```

Plaid_Guest? Yes, it's odd. MOOs assign guests by different colors or some other physical property, depending on the MOO. You may see an Azure_Guest, a Gold_Guest, and many other shades of Guest.

```
*** Connected ***
The Coat Closet
The closet is a dark, cramped space. It appears to be very crowded in here;
 you keep bumping into what feels like coats, boots, and other people
 (apparently sleeping). One useful thing that you've discovered in your
 bumbling about is a metal doorknob set at waist level into what might be a
 door.
You hear a quiet popping sound; Yellow_Guest has disconnected.
Gator opens the closet door and leaves, closing it behind himself.
Rosy_Guest teleports out.
```

Now that you are connected as a guest, use the command @request *<character-name>* for *<email-address>* to request a new character. Because the MOO checks to see if your e-mail address matches the address from which you are connecting, be sure to use a real e-mail address. If the addresses do not match, the MOO prompts you to explain yourself. The next command shown in the following session demonstrates the formula or method that LambdaMOO uses to create new characters. Expect a wait for your new LambdaMOO character. The details of this waiting list mechanism are shown in the following MOO session under Minimal Population Growth (#75104).

Following is a sample of what it looks like when you create a character on a MOO that allows automatic character creation (rather than the delay procedure used on LambdaMOO). This session is taken from Jay's House MOO at jhm.ccs.neu.edu 1709.

```
@request Tarod for busey@eden.com
**** Connecting to Mail server localhost port 25.
*** Mail sent successfully.
...#19
A character named "Tarod" has been created for you. The password has been
 e-mailed to your account. Have a nice day.
```

Now back to the original MOO session!

```
read *b:mpg
The Coat Closet
The closet is a dark, cramped space. It appears to be very crowded in here;
 you keep bumping into what feels like coats, boots, and other people
 (apparently sleeping). One useful thing that you've discovered in your
 bumbling about is a metal doorknob set at waist level into what might be a
 door.
Copper_Guest comes home.
                      Minimal Population Growth (#75104)
                      ----------------------------------
                               by legba (#26603)
            [Last edited on Wednesday, March 16, 1994 at 12:24 pm]

BACKGROUND
This petition is proposed as an alternative to *p:zpg. Though *p:zpg
 addresses a very real social and technical problem, the solution it proposes
 is quite drastic, and it lacks implementation details.

 According to current, very rough estimates, approximately 50 new player
 requests are coming in per day, or 1500 per month, and inactive players are
 being reaped at the rate of perhaps 60-80 per month. Though the figures are
 alarming, there is no way yet of determining whether this is a continuing
 trend. It _is_ clearly advisable to put some mechanism in place to curb the
 growth-rate.

PROPOSAL:
That a waitlist be established from which all new character requests are
 granted that holds up to, but no more than, 500 names. New requests would be
 time-stamped with the day and time of the request, and be ordered
 chronologically.

The number of new players created would be restricted to 5 per day
 (approximately twice what the current reap rate would allow).

Requests would
 be created from the waitlist on a first-come, first-served basis. A
 requestor's name would remain on the waitlist for 30 days, after
which their
 name would be dropped. As vacancies come available from reaping,
requests
 over and above the 5 per day could also be created from the waitlist.

All requests would be assigned a number instead of a name, so that desired
 character names won't be tied up during the waiting period. Once new
 characters are made, they can be given instructions on how to @rename
 themselves.
```

```
A requestor is allowed to update their request and renew interest by some
 @rerequest mechanism, whereupon the date of the original request remains, but
 the expiration is reset for another 30 days. Renewed requests can be made
 repeatedly.

Players who request characters but never log in should be reaped after 30 days.
The number limit of 5 characters per day, the limit of 500 on the waitlist,
 and the time-periods of expiration can be regarded as arbitrary, and
 adjustable by the wizards at their discretion, without submitting the changes
 to the petitions process. If this occurs, however, public notification
 should be provided, with reason given for the change.

(You finish reading petition #75104, *B:mpg)

** Type 'impl *B:mpg' to see the wizards' implementation notes for this
 proposal.
There is new news. Type 'news' to read all news or 'news new' to read just
 new news.
Type '@tutorial' for an introduction to basic MOOing. If you have not already
 done so, please type 'help manners' and read the text carefully. It outlines
 the community standard of conduct, which each player is expected to follow
 while in LambdaMOO.
Fongul teleports in.

help manners

Copper_Guest opens the closet door and leaves, closing it behind itself.
You hear a quiet popping sound; Heartbeat has disconnected.
```

The `help manners` command on LambdaMOO provides a very useful set of guidelines that should be followed by all MUDders. Its output is included here because everyone who plans to use LambdaMOO or any other MUD needs to read it.

```
LambdaMOO, like other MUDs, is a social community; it is populated by real
 people interacting through the computer network. Like members of other
 communities, the inhabitants of LambdaMOO have certain expectations about the
 behavior of members and visitors. This article lays out a system of rules of
 courteous behavior, or "manners", which has been agreed upon by popular vote.

First of all, any action that threatens the functional integrity of the MOO,
 or might cause legal trouble for the MOO's supporters, will get the player
 responsible thrown off by the wizards. If you find a loophole or bug in the
 core, report it to a wizard without attempting to take advantage of it.
 Cracking falls outside the realm of manners. Read 'help cracking' for more
 information.

Beyond that, there are two basic principles of friendly MOOing: let the MOO
 function and don't abuse other players.

====   LET THE MOO FUNCTION   =====
Besides not trying to hack or break things, this means not hogging resources
 by taking up more memory or processing time than necessary.
```

```
To help keep database bloat down, please @create thoughtfully, @recycle unused
 objects, @rmmail when done with it, use feature objects instead of copying
 lots of verbs, and don't recycle and recreate objects seeking "interesting"
 numbers (this inflates all the object #'s, which are long enough already).

The MOO server is carefully shared among all the connected players so that
 everyone gets a chance to execute their commands. The more demanding
 players' commands are, the more of a load there is on the server, and thus
 the more lag there is.

If you are writing a program that will run for a long time, please make it
 wait at least five seconds between iterations (use 'fork (n)' or 'suspend(n)'
 where 'n' is at least 5). This will give others a chance to get their
 commands in between yours.

==== DON'T ABUSE OTHER PLAYERS =====
The MOO is a fun place to socialize, program, and play as long as people are
 polite to each other. Rudeness and harassment make LambdaMOO less pleasant
 for everyone. Do not harass or abuse other players, using any tactic
 including:

* Spamming (filling their screen with unwanted text)

* Teleporting them or their objects without consent

* Emoted violence or obscenities

* Shouting (sending a message to all connected players)
  Don't shout unless you have something everyone needs to hear. This
 basically means emergency system messages from wizards.

* Spoofing (causing messages to appear that are not attributed to your
 character)
  Spoofs can be funny and expressive when used with forethought. If you
 spoof, use a polite version than announces itself as a spoof promptly, and
 use it sparingly. See 'help spoofing' for more information.

* Spying
  Don't create or use spying devices. If you reset your teleport message,
 make sure it is set to something, so that you don't teleport silently.
 Besides having a disorienting effect on people, silent teleportation is a
 form of spying.

* Sexual harassment (particularly involving unsolicited acts which simulate
 rape against unwilling participants)
  Such behavior is not tolerated by the LambdaMOO community. A single
 incidence of such an act may, as a consequence of due process, result in
 permanent expulsion from LambdaMOO.

In general, respect other players' privacy and their right to control their
 own objects, including the right to decide who may enter or remain in their
 rooms.

Also respect other players' sensibilities. MOO inhabitants and visitors come
 from a wide range of cultural backgrounds both in the U.S. and abroad, and
 have varying ideas of what constitutes offensive speech or descriptions.
 Please keep text that other players can casually run across as free of
 potentially-offensive material as you can. If you want to build objects or
 areas that are likely to offend some segment of the community, please give
 sufficient warning to casual explorers so that they can choose to avoid those
 objects or areas.


===== SELF-DEFENSE ======
Avoid revenge!

If someone is bothering you, you have several options. The appropriate first
 step is usually to ask them to stop.

If this fails, and avoiding the person is insufficient, useful verbs include
 @gag, @refuse, and @eject. Help is available on all of these.

If you have a serious problem with another player, you may want to consider
 invoking arbitration, in which another player decides the dispute. Since
 arbitration is some trouble and is binding on both parties, make sure you
 really want it before invoking it. See 'help arbitration' for details.

==== PROBLEMS WITH GUESTS =====
If you are having a problem with someone logged in as a Guest, you have
 another recourse: you may @boot them. Type

        @boot <guest-name>

This will ask you for a reason. Enter the reason on multiple lines, followed
 by a '.' on a separate line. Please note that abuse of guest-booting is quite
 serious, and are subject to the arbitration process. All guest-bootings are
 logged.

=================================================================
If you have a question about something in this text, or about anything else on
 the MOO, type 'help' to see a listing of available help texts. If you don't
 see what you're looking for, page Help or use the Helpful Person Finder in
 the Living Room to find someone who can answer your questions.

If you couldn't read the above text because it scrolled off your screen and
you don't have any text capture mechanism available on your host, type 'help
@pagelength' and 'help @linelength' to learn how MOO can help you read this
and other lengthy text.
```

You will want to do the following as soon as you login with your first character:

```
@gender male
Gender set to "male".
Your pronouns:  he, him, his, his, himself, He, Him, His, His, Himself
```

To define your character's sex and to describe your character, use the following:

```
@describe me as <the message you want people to see when they look at you>

@describe me as Tarod is a striking man. He is very tall and has pitch black
hair. His eyes seem to glow with a brilliant green.
Description set.
```

This will have the following effect:

```
look at me
Tarod is a striking man. He is very tall and has pitch black hair. His eyes
seem
 to glow with a brilliant green.
He is awake and looks alert.
look at emma
tall, quiet and clumsy. short brown hair, wire-rimmed glasses. terribly domes-
tic
 at times, but a music scholar at heart... too sweet for her own good.
She is awake and looks alert.
Carrying:
 emma's nametag                          Emma's membership button
 a cat                                   an alarm clock
```

Your character now has personality and form, so people don't just see white space when they look at him or her. Now you're on your way.

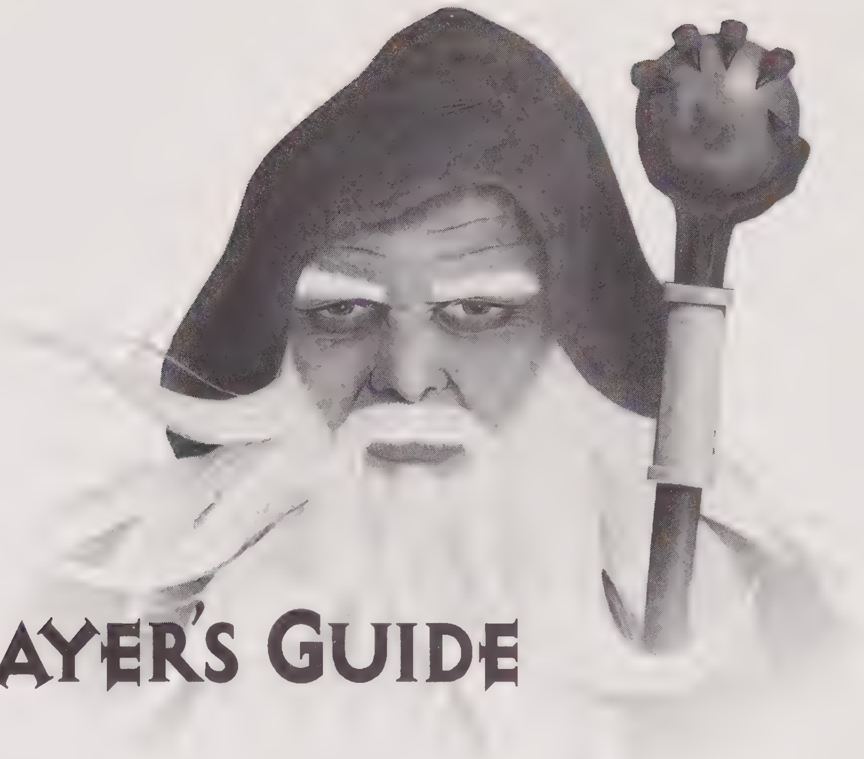
Summary

As you can see from this chapter, MUDs come in many varieties. This chapter walked you through the process of connecting to MUDs and some of the specific idiosyncrasies of the different types of MUDs. The login process probably is one of the ways MUDs are most diverse, so take heart and know that using the different kinds of MUDs will not always be this difficult.

This chapter also should have given you a better feel for how MUDs look and work. Now that you've made it through the basics, go on to Chapter 5, which introduces you to the social commands that are available on MUDs. Soon you will be MUDding like a pro!

II

PART



MUD Player's Guide

5 CHAPTER

MUD Social Issues

The most important part of MUDding is the interaction between the players. In fact, there are several MUDs that are exclusively social environments—they provide neither a role-playing framework nor a game system. However, no matter what kinds of MUDs you choose to play on, you still need to be aware of certain social issues. This and the next chapter discuss MUD social commands, MUD etiquette, and MUD relationships.

Socializing on MUDs

When playing on MUDs, if you don't know something, the best way to get an answer is to ask someone—just like in real life. In real life, however, you don't always have someone there to ask. With MUDs, there's always someone there; but before you can ask someone for help, you need to know how to communicate properly on the MUD.

WARNING

When you are talking on a MUD, the person you are talking to or someone else in the room may be recording the conversation. There also is a chance that a wizard may be watching your conversation. Although it's not common, you need to be careful about what you talk about and who you talk with on MUDs.

Most Windows and Macintosh-based terminal programs and telnet applications enable the user to capture to file everything that is seen. This makes it exceedingly easy to get a perfect image of any conversation that takes place and to later distribute it.

Although it is rare that someone posts a captured conversation on a MUD or on a newsgroup, it is always possible. Some people routinely capture everything they do on a MUD in a file.

Beware of this, and use caution, but also know that most MUDs take care to ensure that the wizards to do not abuse their powers and spy on players. Publishing a captured conversation without the other party's consent is highly frowned upon and could get you ostracized from your favorite MUD(s).

If you decide to capture conversations, I recommend that you only use them as a private reference or as evidence in the rare chance that you are harassed or threatened.

MUDs are fun, and this warning is given as just that—a warning. Relax and have fun, but as you would in real life, be careful who you talk to and what you reveal.

Warnings

It's time to issue another warning: On many MUDs, everything is not as it appears. I have already touched on the fact that many players alter their genders on MUDs (and in cyberspace in general), but that is far from the worst kind of deception that can take place on a MUD.

Recently, articles have begun to surface about all sorts of strange Net happenings: 50-year-old men seducing 14-year-old girls over the Net and meeting them in person; elaborate death threats and plotting. Although these are far from the norm, they *can* happen.

Also be aware that on MUDs, people are not necessarily what they appear to be. This may be because they have a MUD persona that they are living out online, but it may be because they have the intent to deceive. I don't want to scare you from making friends or even developing relationships with people online, but it is important to be aware of what can happen. Metaverse, a MOO run by Steve Jackson Games (Illuminati Online), says:

> Whether you're a kid or not, remember... there are all kinds of people online, and some of them are NOT your friends. Don't assume you can trust everybody you meet in the Metaverse.... If you don't know somebody personally, remember: They might not be the age they claim to be—or the sex they claim to be. They might be a confidence man. They might be a federal agent for any country in the world. They might be your loony ex-husband. So don't tell anybody something you wouldn't tell a stranger on the phone. Okay? Okay.

This is a great warning. Make sure you pay attention to its message. But remember, you still can have fun—just exercise caution and common sense.

Netiquette

Before you go online, you need to learn about *Netiquette*—the unofficial rules of cyberspace:

- **Be nice to other people.** Although this seems painfully obvious, it often is forgotten. It is very easy to forget common courtesies in cyberspace. Respect other people's feelings—people can get upset or hurt by things that happen online just as they can with things that happen in real life.
- **In cyberspace, other people are real, too.** This is another common-sense rule, but remember it. Beyond being nice, remember that everyone you interact with is a real, live person. Treat people with the dignity and restraint you would show them face to face. I once had someone online threaten to kill me. While I didn't take this as seriously as I might have because it was online, the threat never should have been issued. I know the person would not have threatened to my face to kill me. Respect people's beliefs and wishes online—just as you would with a "real person."
- **If you wouldn't think of doing it in real life, then don't do it in cyberspace.** The death threat I mentioned earlier certainly falls into this category. Another example that I have seen only rarely is online rape. Not quite the same as real rape, online rape is more of a very explicit form of sexual harassment. In my years of MUDding, I have never encountered an extreme case of sexual harassment (one that overlaps into real life), but I have seen some pretty vicious and explicit cases of online sexual harassment. This harassment includes continuing lewd comments through `tells` or `pages` after the person being harassed has left the room or asked you to stop. In real life, you would never walk up to someone you don't know and start kissing them or taking off their clothes. You wouldn't do these things in person, and you shouldn't do them on a MUD, either.
- **Help others if you can.** On MUDs, and on the Net in general, you will find that there are always people who know things you don't know. Perhaps you know things they don't know, as well; or perhaps you are new to it all and lost in the confusion. If you have knowledge to dispense or if you can help someone new get started, do it. Someone probably helped you when you were new, and it is only polite to return the favor to someone else.
- **Restrict your use of profanity.** This is a common courtesy because some people are offended by the use of profanity. It is especially important not to use profanity on public channels, such as through the `shout` command. It's also important not to use profanity in messages left on bulletin boards or other areas of public consumption. Remember that on the Internet, it is very hard, if not impossible, to tell the difference between a 12-year-old and a 60-year-old. Respect others if they request that you refrain from using certain offensive phrases.

The Golden Rule of MUDding

Treat others as you would like them to treat you. You will get a lot farther in the MUD world if you make an effort to follow this rule.

Here are some more rules of Netiquette and politeness that are more specific to MUDs (as opposed to the Net as a whole):

- **No spamming.** *Spamming* is the use of macros, clients, robots, or some other mechanism to constantly send a message or set of messages to someone else. It is pretty easy to set up your computer to `spam`, but it is very rude. For example: `Evilone tells you, "You are a jerk!"` This message repeated 40 or 50 times is an example of Evilone using the `tell` command to `spam` you. Spamming in this way often has the effect of causing one's telnet program to choke, which results in being disconnected from the MUD. On combat MUDs, spamming can sometimes result in death. If you are fighting a creature, for example, and the incoming message becomes so distracting or slows you down so much, you might make a mistake or die before you can react. Spamming someone on a combat MUD will certainly make you an enemy.
- **Get permission before using teleport.** This rule goes both ways—get permission before you teleport someone to you or before you teleport yourself to them. You may not be able to do teleport on all MUDs, but many allow teleporting someone to you or teleporting yourself to someone. Ask permission before teleporting to someone so that you don't barge in on a private conversation. Let someone know before you teleport them to you to make sure they have a chance to finish what they are doing. On combat MUDs, you may not get this courtesy if someone teleports in to kill your character (just to let you know ;-)).

Teleportation

If you haven't been around science fiction or fantasy before, you might need teleportation to be defined. Teleportation is the act of moving someone or something instantaneously from one location to another. Some common examples of teleportation that you might be familiar with are the pods used in *The Fly* and *beaming up* in *Star Trek*. Teleportation is accomplished through very advanced technology (science fiction) or magic (fantasy) and is a common power on MUDs. Players sometimes have access to some form of teleportation and virtually all wizards have the power to teleport. Teleporting also is known as *gating* (opening a magical gate between you and your destination). On MUDs, there usually are different powers that allow teleporting to another location or teleporting another player character to your location.

- **Use smileys to identify sarcasm.** It often is difficult to recognize sarcasm and snide comments on a MUD because it is impossible to see facial expressions and body language. So if you are using sarcasm (which is quite common on MUDs) or making a comment that could be interpreted the wrong way, use a smiley to indicate its humorous intent. A smiley is the `:-)` or `:)` (turn your head sideways to see it!). `;-)` or `;)` are winking smileys and tend to mean it's all in fun.
- **No spoofing.** *Spoofing* is the use of clever techniques to send messages that are attributed to others. If you fake a `tell` from someone, this is spoofing. It's difficult to figure out how to spoof; even if you do, you should avoid spoofing.

What Should I Not Do in Terms of Player Interaction?

You shouldn't do anything that you wouldn't do in real life, even if the world is a fantasy world. The important thing to remember is that it's the fantasy world of possibly hundreds of people, and not just yours in particular. There's a human being on the other side of each and every wire! Always remember that you may meet these people some day, and they may break your nose. People who treat others badly gradually build up bad reputations and eventually receive the NO FUN Stamp of Disapproval. The jury is still out on whether MUDding is "just a game" or "an extension of real life with game-like qualities," but either way, treat it with care.

The Communications Commands

Different MUDs have a wide selection of communications options because many are designed solely for interactive communications. MUDs serve many purposes—from theme role-playing to combat to socializing—and they have diverged in many different directions. Because the commands for socializing have been around the longest, they are certainly the most standard—but they are not all the same. I touch on these commands here, and some of the MUD-specific chapters (Chapter 7, 8, and 9) touch on the commands and syntaxes for that specific type of MUD.

After all the discussion throughout the book about how different MUDs are and how unpredictable each MUD may be, it is interesting that the most used command is virtually the same on every MUD I have seen—this is the `say` command. You use the `say` command to communicate with everyone in the room with you.

The following is a sample conversation from a MOO using the `say` command:

```
Doug says, "Tarod, where did you get that name?"
say a series of books
You say, "A series of books."
Kate says, "Oddly enough, I like that song."
```

```
emma [to Doug]: Louise Cooper books!
Doug says, "Who?"
Kate says, "GOOD books!"
say yep!
You say, "Yep!"
Kate says, "Louise Cooper."
emma LOVED those books
"wow! people that recognize the books
You say, "Wow! People that recognize the books!"
```

In the preceding conversation, you can see how the `say` command is used. It is a pretty straightforward command: Use `say` *<message>*, and your message is broadcast to everyone else in the room with you. As you can see in the conversation, I also used `"` as an abbreviation for `say`. The `"` is an abbreviation on MOOs, MUCKs, and MUSHes. On LPMUDs and DikuMUDs, the abbreviation for say is `'` but is used in the same way as `"`.

Many MOOs also have a version of `say` that can be used to direct what you say to specific people.

Use the grave accent (`'`) or hyphen (`-`) followed by the name of the character you want to talk to.

```
'<name> <message>
'"<name1> <name2>" <message>
-<name> <message>
'"<name1> <name2>" <message>
```

```
'Kate, what's up?
You [to Kate]: What's up?
Kate [to you]: Nothing much, you?
-kate not much, just messing around
You [to Kate]: Not much, just messing around.
Kate [to you]: Messing around onna net? It's fun.
-"kate emma" just experimenting with some of these MOO commands
You [to Kate and Emma]: Just experimenting with some of these MOO commands.
```

Emotions

Three important commands for expressing yourself are `emote` (on LPMUDs and DikuMUDs), `act` (on MOOs), and `pose` (on MUCKs and MUSHes). These are different commands that have the same function, so for brevity I will address them collectively as `emote` for the rest of the chapter, unless it is a MUD-specific example. An `emote` is the best way to express your emotions online. The `emote` command often is abbreviated with a colon (`:`) to make using it a little easier. In fact, on some systems, only the `:` works, and the actual `emote` command does not.

```
:smiles.
Tarod smiles.
emote winks at you.
Tarod winks at you.
```

When you use `emote`, everyone sees the same thing. Even if five people are in the room with me in the preceding example, they all see me wink at them (they see `Tarod winks at you.`). This is one of the major drawbacks of the `emote` command. Unfortunately, on several types of MUDs—specifically MOOs, MUSHes, and MUCKs—`emote` is the only way of expressing yourself without using another object that gives you some new commands.

Some LPMUDs do not allow the `emote` command or require that you purchase an emoter (a special object that gives you the capability to `emote`) before you can `emote`. Also, some LPMUDs prepend the output of an `emote` with `*`, `:`, or some other symbol so that the output of the `emote` can be recognized as an `emote`.

```
>:blinks.
Tarod blinks.
```

Others in the room see

```
*Tarod blinks.
```

At first, adding the `*` at the beginning of the `emote`'s output may seem a little silly, but consider the following example:

```
>:gives you 10000 gold.
Tarod gives you 10000 gold.
```

All you see (if the `*` is not used) is

```
Tarod gives you 10000 gold.
```

So you give Tarod your Sword of the Gods, and he runs off. You later wonder where that 10000 gold is, and you realize you were had. This kind of "virtual fraud" isn't a problem on noncombat MUDs because there really isn't anything to lose. On combat MUDs, though, this type of con has resulted in more than a few player killings.

TIP

When you begin playing a new combat MUD, learn what the output of the `emote` command looks like. If the particular MUD you are on doesn't prepend the output with a *, :, or other symbol, be wary of any transactions. You can generally trust people, but check to make sure you get the gold before you give someone the item.

Emotions

LP MUD

DIKU MUD

LPMUDs and DikuMUDs have some special options for expressing oneself online that are much nicer than the basic communications options available on MOOs, MUSHes, and MUCKs. MOOs, MUCKs, and MUSHes sometimes add these options as well; for example, in Chapter 7 you will learn how to turn these options on in a MOO. These commands usually are grouped broadly under the title "emotions" or "feelings."

I have said these commands (the "emotions") were much nicer; by this I mean that "emotions" offer a big advantage over the standard `emote` commands. When `emote` is used, as you have already seen, everyone in the room sees the same message. With "emotion" commands, you can direct your expressions to specific characters in the room with you. This provides a broader (and nicer) way of expressing one's self online. It also provides more depth, as it adds perspective to the conversation. It works like this.

```
>smile
You smile happily.
```

Everyone else sees

```
Tarod smiles happily.

>smile raven
You smile at Raven.
```

Raven sees

```
Tarod smiles at you.
```

Everyone else sees

```
Tarod smiles at Raven.
```

The following is a list of the "feelings" from RealmsMUD. This list also is typical of LPMUDs:

```
> help FEELINGS
ack                 admire              agree               aha
ahh                 annoy               apologize           applaud
backhand            baha                bark                bdance
beam                beep                beg                 bleed
blush               boggle              boo                 boot
bootie              bop                 bored               bounce
bow                 breathe             bsigh               bullshit
burp                cackle              caress              cheer
cherry              chew                chirp               choke
chortle             chuckle             clam                clap
comfort             cough               cower               crack
cringe              cry                 cuddle              curtsey
dammit              dance               daydream            die
disagree            duck                duh                 eek
eep                 eh                  explode             faint
fart                fear                finger              flash
flex                flip                flipoff             flirt
flop                fly                 fondle              freeze
french              frown               fume                gasp
gibber              giggle              glare               gloat
goo                 goose               grab                grimace
grin                groan               grope               grovel
growl               grumble             guffaw              hair
handkiss            harumph             hee                 heh
hi5                 hiccup              hmm                 hold
hop                 howl                hug                 huggle
ignore              insult              kick                kiss
laugh               ld                  level               lick
love                mgrin               moan                mock
nibble              nod                 nudge               ouch
panic               pat                 peer                pfft
pinch               pizza               point               poke
ponder              pounce              pout                puke
punch               purr                puzzle              raise
recoil              roll                ruffle              scratch
scream              shake               shh                 shiver
shrug               shudder             sigh                sing
slap                smile               smirk               snap
snarl               sneeze              snicker             sniff
snore               snuggle             sob                 spank
spit                squeeze             stare               start
steam               stretch             strut               sulk
swim                tackle              tahdah              tap
taunt               thank               think               tickle
tsk                 twiddle             wave                whee
whine               whistle             wiggle              wink
worship             yawn                yeah                yodel
yuck
```

As you can see, there is quite a variety of emotions on this MUD. You will find most of the basic emotions on every LPMUD and DikuMUD you visit, and it's likely that each MUD will have some custom emotions that have been added over time at the request of the players. You may notice that certain commands look a little different on different MUDs (different in the way they appear when you use them). Try them out—they're a lot of fun! You may also notice that everyone seems to have a pet feeling or two that they use a lot—I favor `smirk` and `ruffle`.

tell or page

DIKU MUD

LP MUD

You use `tell` (on LPMUDs and DikuMUDs) or `page` (on MOOs, MUSHes, and MUCKs) to privately communicate with a specific individual, no matter where he or she may happen to be on the MUD. Again, these two commands function the same, but have different syntax on different MUDs so I will use `tell` as the generic reference to this type of command. The player you are talking to does not have to be in the same room as you.

```
tell <player> <message>
page <player> <message>
page <player>=<message>
```

The `tell` and `page` commands are very useful to carry on a private conversation without anyone else knowing it's going on. The following examples show you what `tell` and `page` look like—from the sending end and the receiving end.

On an LPMUD or a DikuMUD, a `page` may look like this:

```
> tell meaglin hi
You tell Meaglin, 'hi'
Maeglin tells you, 'hello'
```

A `page` on a MOO may look like this:

```
page jay hi
Your message has been sent to Jay.
Jay pages, "hello"
```

It is important to remember that in the two preceding examples, no one else sees anything: no one in the room with you or with the recipient sees a message. If you want other people to know that you are talking—perhaps to annoy them or for some other reason—you can often user `whisper`. On an LPMUD, for example, if someone in the room with you `whispers` to someone else, you see

```
Lancer whispers something to Sliver.
```

Other than with the preceding message and the fact that you can only `whisper` to those in the room with you, `whisper` is the same as `tell`. The `whisper` command uses different syntax on different MUDs:

```
whisper <player> <message>
whisper <message> to <player> or whisper "<message>" to <player>
whisper <player>=<message>
```

whisper and _tell_ in Role-Playing

On MUDs where role-playing is an important and integral part of the MUD, you may find an intangible, but accepted, distinction between the `whisper` and `tell` (or `page`) commands. You use `whisper` for in-character conversations (when you are telling your associate in the same room that you think another character in the room with you is lying, for example). `tell`, on the other hand, is used for out-of-character conversations (when you ask your associate his real name and where he went to college, for example). This honor system helps keep the game and real world separate, and keeps people from exchanging game information in ways that their characters could not.

LP MUD

Another important distinction between `whisper` and `tell` comes only on LPMUDs and is due to a restriction placed on `tells`. As touched on briefly in Chapter 3, LPMUDs charge characters "spell points" for using `tells`. This practice helps prevent spamming via `tells`. If someone `tells` or `pages` you with an annoying message a hundred times in a row, it can be pretty annoying—and if you are in the midst of fighting a powerful monster, it can be deadly.

Shouting

Most MUDs have `shout` lines or other forms of gossip-oriented communication. `shout` is the most common command of this type and it sends a general message to everyone currently on the MUD. For example,

```
shout <message>
```

sends `<message>` to everyone logged on to the MUD. Other commands similar to `shout` include `gossip` and `chat` (for basic talk), `auction` (for selling items to other players), and `gratz` (for congratulating other players). Some MUDs even have more `chat` lines called *channels* for things like talking to everyone who is your race (for example, all the elves), everyone in your class (all the priests), and many others. The `shout`-like commands will vary from MUD to MUD, but they all work the in the same way as the `shout` command discussed here.

You shouldn't have much trouble recognizing these global messages when you start to see them. Shortly after you start seeing them, if you are on a very social MUD, you will probably want to figure out how to turn some of them off. This is very MUD-specific—some MUDs don't allow you to turn these messages off. If you ask one or two experienced players on the MUD, you can get a quick idea of how to turn these global messages off and how to identify them.

As a reference (in case you encounter this), wizards on LPMUDs also have a special command called `echoall` that allows the wizards to send any message to the whole MUD (other MUDs have similar commands). This message isn't preceded by the wizard's name, so it appears in the midst of normal activity. Sometimes rowdy (drunk or renegade) wizards can get pretty creative with these messages. I've seen an `echoall` range from the standard LPMUD death message, after which everyone on the MUD freaks out, to a `VICTIM-NAME tells you, 'slanderous comment'`—which can obviously make the person being imitated (the victim, generically portrayed as VICTIM-NAME) new enemies (this is an example of spoofing).

Who

Now that you know how to talk to people, I guess it will be pretty useful to know how to find people to talk to—or at least see if they are on the MUD.

```
who
@who
WHO or +who
WHO
```

The preceding command gives you the list of everyone playing on the MUD. Sometimes the `who` command also includes other information, such as where on the MUD each player is and what level each person is, and some basic information like the player's guild and whether the player is idle or not. The `who` command is incredibly useful for checking to see if your friends are currently on the MUD, to see how populated the MUD is, and to get a feel for who the players are.

The following shows some example lines of what you might see when you use the `who` command:

On an LPMUD

```
(5)     Geran the Cutpurse (Mortal)
```

On a DikuMUD

```
[13 Wa   ] Vile the Human Soldier [SHOGUN Journeyman]
```

On a MOO

```
*Player name      Connected    Idle time    Location
-----            ----         ----         ----
tarod (#99726)   7 seconds    2 seconds    The Coat Closet
```

On a MUSH (using `+who`)

```
  Name                    Sex_Idle_On_For__Location                          (DB)__Class
Dr. Klerk                (M)   4s  00:33  Dark Alley                       #2513
```

On a MUCK

```
Player Name           On For Idle
CrystalCat             00:29   1m
```

As you can see, the `who` command varies widely among the different kinds of MUDs. It also varies widely among each individual MUD.

Summary

This chapter gives you a good idea of the social commands available on MUDs. With these commands, you can communicate with other players and at the very least, ask for help. You now can see who is on the MUD. This, in tandem with the basic navigation skills you have already learned, will make you a lot more competent than the average newbie. When I started MUDding, I didn't even know what `say` was!

With this, you are, at a minimum, equipped with most of the skills you need to become an active citizen on a social MUD. To learn more about the specifics of individual MUDs, you will want to read about them later in Chapters 7, 8, and 9. You definitely will want to check out Chapter 6 and learn about MUD relationships and romance—one of the more interesting aspects of MUDding.

6

CHAPTER

MUD Relationships

WARNING

This chapter deals with adult topics. You may find some of the discussion in this chapter to be offensive.

Many things on MUDs mirror reality, and others work in ways that are very alien. This chapter relies heavily on personal experience and interviews with other MUD users who have gone through some of the things to be discussed. This chapter contains material that may be offensive to some, but the topics discussed are taken from reality. If you plan to participate in MUDs and never let anything from a MUD enter into your real life, you may want to skip this chapter—but I was also one of the people who said I'll never meet anyone from a MUD in real life. Boy was I wrong.

This chapter is not hard-fact and function oriented like the rest of the book. Most of what I discuss in this chapter is much more subjective, but I think the topics discussed in this chapter are an important part of MUDding and cover something that needs to be discussed. I rely more on personal experience and the experiences of other MUDders for this chapter and less on rules—that's because there are no rules. Some of the narrative even leaves the MUD and enters into real life—some of the things in this chapter really happen. Perhaps reading about these experiences will prepare you for when you encounter similar experiences—or you may just find it all entertaining. Whatever your motives, what I talk about here is a part of MUDding—and for some, it is the most important part.

Some of the things I say in this chapter are discouraging. I talk pretty negatively about online relationships and the real life relationships that may evolve out of them. I have had several online relationships, some of which have been really good. You will recognize the good relationships, which are pretty subjective. Therefore, much of this chapter tells you how to recognize the bad relationships and to give you an idea of what you can expect out of this type of relationship if you get into one.

MUD Friendships

Yes, it's true—there are nonromantic relationships that grow out of MUDs. I have many male and female friends whom I consider close friends. I know these friends primarily through or from MUDs. But these friendships have not always developed the way friendships in real life seem to develop. I have found that the camaraderie that is developed from adventuring, and even just hanging out on MUDs, leads to the development of very strong friendships that can carry over into real life.

In college, I remember many evenings that I spent up late at night talking to people. These were conversations that were much more open and pure than any I had before or any I have had since. Talk spanned over politics to philosophy, and much more. In such conversations, you reveal much of your inner self and develop your ideas as you discuss them. These conversations never seem inhibited—they are invaluable. This is the type of interaction that leads many to say that some of their best lifelong friends are those they've made during college.

I don't know whether this is true for everyone, but it is true for me. I believe that this kind of openness and the extended time spent talking leads to the development of these relationships—and MUDs have this same kind of effect.

As you may have found already, it is not uncommon to spend many consecutive hours on a MUD. I have watched the sun come up many mornings after MUDding all night and have had no grasp of the time passing. This immersion in the MUD environment is the bond that binds MUD friendships together. That's because most of these all-night sessions were spent in a group with several other people wandering around the MUD killing monsters and collecting treasure. On most MUDs, you have to stop and wait for monsters to come back or heal between forays into the various adventuring regions. These down times inevitably lead to conversations about real life, especially if you are in a group you have spent several nights with adventuring.

This is not to say that everyone you adventure with is going to become a lifelong friend. In fact, there are many people on MUDs I talk to when I encounter them on the MUD, but we have no contact outside the MUD. As is the case with friendships based entirely in our reality, these friendships have a high degree of variance in their closeness. Among my "MUD-only" friends, there are many who know nothing about my real life and whose real life I know nothing about. Other "MUD friends" know something about my real life, and I know something about theirs, and we talk about real life things in the course of conversations on the MUD.

There are also friendships that develop on MUDs and that turn into real life friendships. Not to imply that MUD friends aren't real friends—MUD friends are, but they are not the same as friends with whom I discuss real life issues. By real life friendships, I don't necessarily mean to say that I know the person in real life. I have many friends I talk to on the phone and via e-mail whom I have known for a long time. I consider them as close as I do some of my friends I have met in the real world. I talk to them about real life things. What separates these friends from MUD friends is that I talk to them on the phone and via e-mail—communications channels outside of the MUD. This kind of off-MUD talk tends to make a conversation more real and gives it more weight.

MUD Romance

I've had a wide variety of online romantic relationships. I'm going to provide examples of the different types of relationships that I've seen develop through anecdotal stories. Because I've (embarrassingly enough) been through most of these types of relationships personally, I think I have a good idea about how they work. I used to say, "I'll never meet a girl through a computer." If this sounds like you, you may want to read about these different kinds of MUD romances—whether it's because you may find yourself in a MUD romance or because you may get a good laugh out of it. I've changed names to protect the innocent (or guilty, as the case may be) and have altered some of the stories. In altering the names, I have chosen to substitute names from Greek mythology. It is highly likely that there are real MUDders out there using some of these names, but they are not the players dicussed here. The names from Greek mythology are used so that I had one single source from which to draw names. Some stories are also composites of things that have happened to me and things I have talked about with other people.

The Worst

I'll start with the worst kind of MUD romance. We'll call the two characters Zeus and Hera. I'll tell the story from Zeus's perspective.

It starts innocently enough. Zeus is wandering the MUD happily killing things and ends up forming a party with Hera so that they can go after bigger monsters. Zeus doesn't think much of adventuring with a female character—it's no big deal. So they kill monsters and explore the MUD for a couple of hours. Then both log off and go do real life things.

The next day, Zeus sees her (Hera) online again and says "hi." They form another party and wander off into the MUD. They talk a little more this time but about the MUD. These casual conversations continue for a week or so. Now Zeus and Hera actually are arranging times to meet online so they can adventure together. Zeus logs on to the MUD to see if she is online, and if she isn't, he logs off.

Next thing you know they are MUD married. Whoa! What? Yes, it's true. I said "MUD married." Learn more about this concept by reading the "Mud Marriage" sidebar.

MUD Marriage

A MUD marriage is when two *characters* on a MUD get married on the MUD. This does not mean there is a relationship in real life; the marriage is exclusively between the two *characters*. The concept of MUD marriage is very alien to many people. Keep in mind that MUD marriage isn't anything like the real life version of marriage. Even within the MUD, the characters aren't quite as devoted to the marriage, and they don't take it anywhere near as seriously as most people take real marriage. People often propose MUD marriage within a week or two of meeting (another example of the hyper-relationships that often develop on MUDs). So don't be surprised if someone proposes to you after a night of adventuring. On MUDs that have an orientation toward more serious role-playing, however, there may be months of friendship and flirting ("MUD dating") before that special MUD someone proposes to your character.

In fact, some players even have multiple characters on the same MUD that are married to different characters. Loyalty is a little bit different in MUD marriages than it is in real life. In fact, I know several people who are married in real life and also on MUDs, and not necessarily to their real life partners.

Now some wives (or husbands, girlfriends, or boyfriends) may take their significant other's having a MUD spouse with a grain of salt and write it off as only a game—and most of the time, it is only a game. But I would strongly recommend letting your real life significant other know because I've seen some pretty crazy things happen in these types of situations.

So keep in mind that if you get MUD married, it probably doesn't mean all that much—it's a marriage between the two characters and not between the two people. It is generally used as a mechanism for role-playing (the characters are "in love") or just as a symbolic thing. Occasionally, a MUD marriage is for convenience—so that one of the two doesn't get hit on by a bunch of strangers all of the time.

So now Zeus and Hera are "MUD married." This marriage really doesn't change much except that other characters now look at Zeus and Hera as off limits. The MUD marriage probably didn't change much for the characters. However, I tend to want to know at least a little something about someone before I MUD marry them, so let's say that at this point, Zeus and Hera have introduced themselves with their real first names.

From here things can diverge. The simple path is that the relationship remains where it is. Zeus and Hera are MUD married, and they spend a lot of time together on the MUD. Maybe they talk a little about real life, but not too much. The other path is the more interesting path, which leads us to the worst kind of MUD romance.

Now Zeus and Hera (or one of them, anyway) decide they are interested in each other in a more-than-MUD way. So they begin talking about more than MUD things or begin doing things that aren't really just "friendly"—for example, MUD sex.

Having MUD sex can be a meaningless, casual thing that is done for fun, or it can be something taken seriously inside the confines of MUD marriage. MUD sex can be between two real life lovers who are hundreds of miles apart (perhaps they met on a MUD). Zeus and Hera obviously aren't just having casual MUD sex because they are closer than just MUD friends.

MUD Sex

MUD sex is another MUD item that may seem a bit shocking to some. MUD sex (sometimes called TinySex—usually on TinyMUDs, MUCKs, and MUSHes) is a lot like phone sex. As you know, most MUDs have a high degree of flexibility when it comes to expressing oneself and communicating—and if you're a little creative, you can use these commands (such as `say` and `emote` discussed in Chapter 5) to have MUD sex (or TinySex, depending on the type of MUD it is).

In fact, some MUDs have gained a reputation for being a good place to go if you want to have MUD sex with a character—kind of like Internet pick-up spots. FurryMUCK, discussed in Chapter 3—remember the anthropomorphic animals?—has this reputation. MUD sex and "picking up players" is not an uncommon theme on many LPMUDs.

The MUD FAQ has some interesting things to say about MUD sex. Notice the description of *logs*—you may want to make sure that you trust whomever you have MUD sex with not to embarrass you.

What Is a Log?

Certain client programs allow logs of what you see on-screen to be kept. A time-worn and somewhat unfriendly trick is to entice someone into having TinySex with you, log the proceedings, and post them to `rec.games.mud.*` and have a good laugh at the other person's expense. Logs are useful for recording interesting or useful information or conversations, as well.

What Is TinySex?

TinySex is the act of performing MUD actions to imitate having sex with another character, usually consentingly, sometimes with one hand on the keyboard, sometimes with two. Basically, it's speed-writing interactive erotica. Realize that the other party is not obligated to be anything like he/she says, and in fact may be playing a joke on you (see the preceding explanation about Logs).

Back to Zeus and Hera—their first step out of the world of casual relationships was MUD marriage. But that doesn't generally mean much, and Zeus did it more for the experience than anything else. So next, Hera seduces Zeus (in a purely MUD sense) into having MUD sex. He finds it entertaining, so they start having MUD sex on a regular basis. Soon they are talking a lot more, and they are even seeing each other on different MUDs.

At this point, Zeus is beginning to feel a little awkward because he wasn't really prepared for all this. Hera is talking about real life things and seems to be taking this MUD sex stuff pretty seriously. So what does Zeus do to alleviate his worries? He asks to talk to her on the phone. Hera is hesitant to respond to this request, claiming that her voice sounds bad and that she would be ashamed. This hesitation only makes Zeus more suspicious, so he becomes very persistent. Hera continues to resist until Zeus issues the ultimatum: Talk to me on the phone, or it's all over.

So Zeus gives Hera his phone number, and she promises to call him collect. Zeus anxiously waits for the call. When the call finally comes, it only takes Hera's first word for Zeus to come to the horrible realization that Hera is a guy. (This does not necessarily mean to imply that Hera is homosexual. He could have done this as part of some sort of cruel joke, to get something out of Zeus, or just as some sort of exploration.)

Are There Limits to the Deception?

You saw how the relationship developed between Zeus and Hera in the preceding section. The sequence of events that brought them together is not uncommon. In the following examples, the people being discussed could have had their relationships develop the same way as Zeus and Hera, but with a different ending. Keep that fact in mind when reading these other examples.

Athena meets Poseidon through some friends who always hang out together on the MUD. Although Poseidon doesn't know any of them in real life, some of Poseidon's friends have played with Athena on the MUD for some time. Athena gets kind of bored in real life and starts playing MUDs more. She is having problems with her real life boyfriend and begins to retreat into the MUD more than usual.

Athena and Poseidon adventure together, but really aren't too grounded in the MUD. They talk via e-mail very early and quickly begin exchanging real life letters. Talking on the phone also happens quickly for Athena and Poseidon. Poseidon is quick to say all the things Athena wants to hear about her real life problems, and she gets pulled in by him.

Athena asks Poseidon more and more about himself, and they begin to talk a lot about real life. More phone conversations occur, and more letters are exchanged. A month later, Athena thinks she is falling in love with Poseidon in real life. They talk more and decide they should meet in person. They make all the arrangements to meet (Athena will fly to see Poseidon) and wait patiently and talk continually in real life—at this point Athena believes she is in love with Poseidon.

So with great expectations, Athena gets on a plane and flies across the country to meet Poseidon in real life. Athena gets off the plane and is greeted by Poseidon. Upon seeing him, she realizes that it has all been an illusion. Most of what Poseidon has said about himself has turned out to be exaggeration or fantasy. Upon arriving at his apartment, the visit quickly becomes worse as Poseidon exhibits none of the social skills he showed over the phone and on the MUD. He appears to have no depth or even the appearance of a life —he is completely different than the image he painted over the phone and via e-mail.

WARNING

What Athena now does is unwise. If you are a female MUDder, I highly recommend that if you decide to meet a guy from online in person, you make the guy come to see you. Nothing can ever be completely safe, but at least this way you're on your home turf and have friends and places to go if your meeting with Mr. Right goes awry. If you have just flown a thousand miles to meet someone and it doesn't work out, you're stuck with this person until you can get home. While guys certainly take a risk by flying out to meet a girl they have met online, girls take the larger risk if they put themselves into a situation with a guy they don't really know. I have never heard of a rape or anything incredibly bad coming out of such arrangements, but with the growth of the Internet, I think it should be a concern.

Athena gets the next plane out and is gone after spending only one night of a week-long trip. The two never speak again.

A Better Situation

It's time to introduce a couple more people. Ares and Aphrodite meet online and start hanging out very early. They never really go adventuring together—they just sit around online and talk. They also start having MUD sex very early on, and it is a pretty torrid relationship. This example is not uncommon—people hit on each other on MUDs just like in real life, and relationships exist that have a purely (or mostly) sexual foundation. As Ares and Aphrodite continue to meet online and have MUD sex, things become more intimate, and they start to talk on the phone as well as on the MUD.

What happens next is also very common with MUD relationships. If you have MUD sex with someone, there will likely be some pretty vivid exchanges—and if it is serious, it can be pretty intimate. And hey, if you're willing to have MUD sex, phone sex doesn't seem like such a big jump. Ares and Aphrodite made that jump. The phone conversations between Ares and Aphrodite made their relationship much more intimate.

Not surprisingly, Ares thinks they should meet in real life, so he asks her. She agrees, and they set a date to meet at her university. He'll fly in, she'll pick him up, and they'll spend five days together. So they plan far in advance, around spring break—which turns out to be about two or three months away. So they continue to talk, and the anticipation builds as the time of their real life meeting approaches.

But in the months between the meeting and the time it is arranged, things don't go entirely as planned. Aphrodite starts seeing someone in real life, and the relationship between Ares and Aphrodite begins to weaken. But they decide that they may as well go for it and meet anyway. They've been through this much—may as well see it to the end. So they deal with the things that happen up until the meeting.

They meet in the airport and immediately recognize each other from physical descriptions and what they said they'd be wearing. After picking him up, they ride back to her university. As you can imagine, the first meeting is awkward. Neither one has ever done this before (meet someone from a MUD in real life—specifically someone they've had this sort of relationship with), and it's weird. There's a lot of emotional build up, but both are uncertain what will really happen.

That night they arrive at her dorm and talk for a while. Then they have dinner and return to her dorm. Things have become a little more relaxed, but neither one seems to really know how to react. So after talking for a while, it's time to go to sleep. Because her room is a single, the room has only one bed, thus leaving them in a potentially awkward situation. So she invites him to sleep in her bed.

Nothing happens that night, but there seems to be something there. The next day, things get a little heavier, and they begin kissing. But Aphrodite has to leave for the day to take care of some family business, so Ares spends the day contemplating what's going on. The next day, she returns and they spend the day together. The more time they spend together, the more comfortable they feel with each other.

The next day, it happens. They have sex. This consummation of their relationship may well have been the beginning of the end. It does relieve a lot of the "So we've done all this on the phone and online, does that mean we should do it in real life?" type questions. Although having sex doesn't resolve the sexual questions in the right way, it does resolve them. The next day, some of Aphrodite's real life friends, some of whom have MUDded, arrive to hang out—one of which is the guy she is sort of seeing in real life. Things fall apart from there.

After Ares leaves, they continue to meet and chat online and usually are cordial. But, they never talk again on the phone nor do they send e-mail to each other. The relationship just ends.

Are There Happy Endings?

You're probably starting to get a pretty dismal picture of MUD relationships right about now. They aren't really as bad as they seem. Just like in real life, relationships are often rocky, and every relationship is a learning experience. Because there are different things to learn in MUD relationships, these examples are mostly negative so that you know what kinds of things to expect. I don't think you'll need a lot of help if you're one of the lucky people who finds the right relationship.

As is not uncommon on a MUD, Apollo shouts something about his geographic location one evening. Circe responds that she knows someone who lives there. Then she makes a casual comment about someone Apollo knows on the MUD. So Apollo, being curious, asks if he can call Circe and talk to her off the MUD. Circe accepts.

Apollo and Circe talk that night on the phone. They become fast friends. They speak on the MUD a lot, exchange e-mail every day, and talk on the phone often. They never have MUD sex or phone sex. The relationship is purely platonic. Or at least that's what they both pretend. But they decide to meet and just hang out. But both harbor expectations of something more, although neither readily admits it.

So they meet in real life, and there's a spark. Although they are only together for less than a week, things become very intimate between them. A relationship springs out of that meeting. A month later, Apollo flies out to see her again. From there they meet nearly every month and talk on the phone every day. They don't see other people, and they're in love.

They have the problems of a normal relationship over the next several months, compounded by the problems of a long-distance relationship. Although there are rocky moments, they stay together for nine months. Then Apollo moves to where Circe lives. Finally, they are together.

Well, that's almost happily ever after. They go out for another nine months before they break up. The relationship's end has nothing to do with MUDs. It just runs the course of a normal relationship and ends, but it's one of the happier MUD relationships that I know of.

Things to Remember in MUD Romance

Here are some things to watch out for with MUD romance.

- **Real life involvement.** This is a big warning sign. If the person you are interested in seems to have some real life involvement—either an existing significant other or someone comes along during your relationship—be very careful. Often people will meet on MUDs and talk about problems in their existing relationships. I've watched relationships develop this way, but be leery of them. Sometimes people look for "safe" MUD relationships to escape bad real life relationships.
- **Imbalance in the expression of feelings.** If the person you are "seeing" online says that he or she loves you or shows some equally strong indication of feelings, and you don't feel the same way, problems could be on the way. Or if you feel like you love someone, and he or she never says anything of equal emotional strength back to you, watch out.

- **Make sure the person you are involved with is the gender you think they are.** It seems pretty obvious, but it could cause big problems if you get involved with someone who is pretending to be someone they aren't (refer to the example in the section "The Worst"). Before the relationship escalates into something that spills over into real life, make sure you know the gender of the person you are becoming involved with. A phone call is an easy way to solve this mystery.

Following are some tips to remember:

- **Every computer has a Backspace key.** With a Backspace key, any spontaneous comment can be deleted or re-written before you ever see it. Remember that every `say`, `tell`, and e-mail can be composed (written specifically with some purpose in mind, rather than the spontaneous conversation that you might think it represents). While it may seem spontaneous, it is certainly easy enough to calculate every word that is said or written. So remember, everything you "hear" online could be constructed in this way. Talk to that person on the phone, at least, before you start to build a clear picture of him or her.
- **Be wary of that picture.** If someone sends you a picture (or a computer image) of themselves, remember that it may not be them. It probably is, but you never know for sure. If you plan to meet someone, it is pretty self-defeating to send a fake picture because as soon as they see you, they'll know it's fake and backed by lies and deception—not a good way to start a meeting. While the picture is probably the real person, remember that it is also a chosen picture. It's probably the best picture the person has, so don't get too worked up over a picture.
- **A picture doesn't show everything.** You could madly love someone's personality from talking to them for hours on the phone and on MUDs. You could send e-mail back and forth every day. You could even have a real, accurate picture of the person, and you find them to be physically attractive. Remember that when you meet this person, they could have any number of traits that aren't apparent from pictures or words: they could have body odor, they could pick their nose chronically, they could have obnoxious ticks that drive you nuts.
- **MUD sex doesn't necessarily mean real sex.** After you have had MUD sex or phone sex with someone, it creates a certain expectation of what might happen in real life. If things are serious and MUD or phone sex has occurred, it is logical that one of the two parties may be hoping or expecting real sex when they meet in person. But that isn't always the case. For the reasons discussed previously, or for some other reason, one party may decide that real sex with the other is undesirable. Be prepared to cope with this possibility. In fact, go into the relationship expecting this to be true.

- **Keep your expectations in check.** When getting on the plane to fly out to meet someone or sitting in the airport waiting for them, you might think you already love this person—and you might. But you have to give things time. It is very rare that on first sight everything becomes real. Usually, there is some awkward time, maybe an hour or maybe a few days—or maybe forever. Treat your first meeting like a first date, not a reunion with a long-lost lover.
- **Hyperdevelopment of relationships is not uncommon.** While you may treat your first real life meeting with someone you met online like a first date, a sixth date may be the next day. MUD meetings like this may go from first-date type talk to an intimate relationship in days. When you visit someone on a MUD and are around them for such extended periods of time, things may develop much quicker than they do in real life. A visit of this kind can be a roller coaster ride.

Summary

Now you have some idea of what could happen in MUD-based interpersonal relationships. While some may say this never happens, you never know what might happen when you start talking to people. You could find yourself making a lot of new friends, or even falling into a romance. Hopefully, this chapter has prepared you for some of what you might encounter in the online world. Remember, people that are online can play just as many games as in real life—or they can be just as sincere and passionate.

7

CHAPTER

The Social MUDs—MOOs, MUSHes, and MUCKs

While all MUDs are social, some MUDs have only a social aspect. But, as used in this book, a social MUD is one that does not have a built-in combat system. This usually includes MOOs, MUSHes, and MUCKs. These three obviously do not have MUD in their names, and have lead to the use of the MU* acronym as a generic term for MUDs. *MOO* stands for *MUD Object Oriented*. The social MUDs also tend to have a stronger tendency to allow players to create new objects; however, some restrictions (especially on new rooms) exist.

Another difference that is found on social MUDs is that the players can almost always create objects, and don't need any special powers or rank to create basic objects. Unlike on most combat MUDs, where only wizards (see Chapter 11) are allowed to create objects and rooms, the policy for creating objects on social MUDs is more open. Creating objects is covered in Chapter 14 for MOOs and Chapter 15 for MUSHes and MUCKs.

Different Things to Do on a Social MUD

Even though a social MUD is just a classification, it is somewhat difficult to explain a social MUD. Because social MUDs don't have quite the level of distractions (other than socializing) as do combat MUDs, there must be draws. Following are four attractions of socials MUDs that can be easily identified:

Socializing: Socializing is chatting with other players, hooking up, having MUD sex, or whatever you find interesting. You probably can find someone else who shares an interest with you. It is hard to describe this type of socializing—just jump into the maelstrom and start talking to people.

Building: Building is the capability to create new and interesting objects that do neat things, such as adding new rooms. This "god sense" (the power to expand or create a new world) attracts some players; others just want to express their creativity. Some players just want to add to the world. Because social MUDs allow players to be more creative, whereas combat MUDs require that you must be a wizard before you can create objects, social MUDs tend to have many builders.

Exploring: Exploring, in essence, is watching or looking. Because social MUDs enable players to build objects, there is more to look at than on combat MUDs (where only wizards can add new things to the world). Some players spend days just wandering around, looking at the different rooms and objects and trying to unlock their hidden secrets.

Role-playing: Just because a MUD is not considered a combat MUD, does not mean it cannot have a role-playing focus. Some social MUDs have strong role-playing systems that use existing role-playing games as a base. For example, the *Amber Diceless Role-playing Game* is a role-playing game that does not involve dice and has a system of combat resolution that does not require all the statistics that are used in many other role-playing games. This enables players to develop their characters without worrying so much about combat, but instead focusing on the interaction with other characters and storytelling. Most of these MUDs have rigidly defined themes, for example, confined to the world of a series of books, such as Anne McCaffrey's *Dragonriders of Pern* series. This type of role-playing often is much different than that found on combat MUDs.

So there you have it. You may not fall into only a particular category, but you probably spill into at least one of them.

WARNING

Social MUDs, in particular LambdaMOO, have huge numbers of players logged in simultaneously (LambdaMOO often has more than 200 people logged in at one time). Many players are guests or are hanging out, talking in the starting or a nearby room. This typically is called *noise*. If you enter a room that has 20 players talking, you probably will lose track of what is going on. Watch out for stray noise, and as you get used to it, you can determine your tolerance level. Then when you become overloaded with too many conversations, just invite the players you want to talk with into a private room.

MUSHes and MUCKs

MUSHes and MUCKs, which are in the social MUD category, closely resemble MOOs. All three came from the original TinyMUD system. In fact, the original author of MOO was also the original author of TinyMUCK. Because the basic commands (commands used for communication) already have been detailed (in Chapter 5), they are not discussed here. Chapter 14 addresses programming MUSHes and MUCKs. For a list of basic MUCK and MUSH commands, check out the inside back cover of this book!

MOOs

Although as shown in Chapter 4, you might have to wait a bit to get your new MOO character—especially on Lambda MOO. LambdaMOO is far and away the largest MOO in existence, with more than 200 users at any given time.

Lambda MOO is only one MOO and certainly not the only MOO out there. Also of interest is Jay's House MOO, which is at `jhm.ccs.neu.edu (129.10.111.77) 1709`. Jay's House is more of an experimental MOO where players develop new MOO-related projects. Players also just hang out and talk. Media MOO, sponsored by MIT's Media Lab, is at `microworld.media.mit.edu (18.85.0.48) 8888`. MOOs have many different themes; for example, Media MOO is an academic MOO, whose stated purpose is the following (seen by using `help purpose` on MediaMOO):

```
MediaMOO is a professional community for media researchers. It is a place to
come meet colleagues in media studies and related fields and brainstorm, to
hold colloquia and conferences, and to explore the serious side of this new
medium.

Unlike other MUDs, characters on MediaMOO are identified. You can find out who
anyone is with the @whois command (except for a few early members who are still
anonymous) so that you can contact them to continue professional discussions.

To become a member of MediaMOO you must be doing media research. We are more
interested in knowing about what you are doing than what you are interested in.
Most college and pre-college students who apply are not really doing media
research.

If you're looking for a place to hang out with interesting people, LambdaMOO is
at lambda.parc.xerox.com 8888. In fact, if you read the Usenet newsgroup
rec.games.mud.announce, you'll find a list of hundreds of MUDs and MOOs, almost
all of which have no requirements for membership.

A note to teachers:

Unfortunately, we must discourage you from bringing classes of students here.
Although these experiments are interesting, this is not the appropriate place.
MediaMOO would become a very different place if it were filled with, for
example, hundreds of freshman composition students.
```

```
Welcome!

Sincerely,

Amy Bruckman

September 1993
```

While Media MOO might not be the place for you, it is a good example of how MOOs are being used constructively.

MOO Beginnings

It is easy to assume that on a social MUD, the most important commands would be the social commands. This is not a bad assumption; however, there is a command that rates even higher than the social commands—the `help` command.

`help` by itself gives you a list of basic topics. Help on MOOs also tend to give you a good idea of what you might want help on. If you mistype something, for example, `help` will give you one or two options that are close to what you typed. `help` is a good way to learn the system and resolve any questions you may have. You also can substitute `#<object number>` for `<topic>` if you need help on a specific object. Object numbers (`#`) are introduced a little later in this chapter.

Of course, if you cannot figure out what you need to know using the help system—ask someone.

At this point, I normally would introduce you to the important social commands, such as `say` and `page`. If you have gotten to this point, however, I am assuming that you have picked them up in the previous chapters. If you have skipped ahead to this point and are not familiar with `say` and `page`, refer to Chapter 5.

For convenience, Table 7.1 provides is a brief summary for your reference:

Table 7.1. MUD commands.

Command	*Abbreviation*	*Description*
Communications		
`say <message>`	`"<message>`	Says `<message>` to everyone in your current environment.
	`-<name> <message>`	Says `<message>` to all in your current environment, but is directed to `<name>`.
`emote <message>`	`:<message>`	Emotes the given `<message>` in the

Command	*Abbreviation*	*Description*
		form `Your_name` *`<message>`*.
	`::`*`<message>`*	A special emote in the form of `Your_name`*`<message>`*. (No space.)
`page` *`<player>`* *`<message>`*	`'`*`<message>`*	Sends a private *`<message>`* to *`<player>`* anywhere on the MOO.
`whisper` *`<message>`* `to` *`<player>`*		Whispers a private *`<message>`* to *`<player>`* in the same room.
Player Set-Up		
`@describe me as` *`<description>`*		Other players see `<description>` when they look at you.
`@gender` *`<gender>`*		Sets your character's gender. `<gender>` is most likely `male` or `female`.
`@password` *`<old password>`* *`<new password>`*		Changes your character's password.
Other Useful Commands		
`look` *`<object>`*	`l` *`<object>`*	Looks at *`<object>`* to see its description.
`inventory`	`i`	Sees the objects you currently have in your possession.
`@who` *`<player>`*		Sees who is currently logged on or if a specific *`<player>`* is on.

If you already know how to navigate and play on MOOs, and you want to create objects or program code, look ahead to Chapter 14, where you can learn how to create objects, rooms, and more.

Object Numbers in MOOs

MOOs (as well as MUSHes and MUCKs) are different from the other types of MUDs, in that MOOs use object numbers to keep track of everything in the MOO world. (Technically, the difference is that players can see the object numbers; the numbers themselves are used on most kinds of MUDs.) Every item, room, and player has its own object number. Object numbers designate individual objects, not instances of those objects. The following is an example of the way in which object numbers work using the `@who` command:

```
@who
Player name              Connected        Idle time        Location
-----------              ---------        ---------        --------
Umber_Guest (#714)       10 minutes       0 seconds        The E&L Garden
Cyan_Guest (#531)        30 minutes       a second         The Library
Andrei (#10278)          34 minutes       15 seconds       Public Library
Guest (#113)             38 minutes       32 seconds       Infocenter
```

The `Umber_Guest` is listed with `(#714)` next to the name, which indicates that `714` is his or her object number. Object numbers are very useful on MOOs and you should note relevant numbers for quick reference. It often is much easier to find things by object number rather than through other methods, such as guessing what their "official" name might be.

MOO Navigation

Some MOOs, contrary to some of the MUDs you have already seen, do not always use the cardinal directions.

```
Obvious exits: library to Library Foyer, atrium to Third Floor Atrium Landing, and
common to Curtis Common
```

As you can see from the preceding line, the exit commands are `library`, `atrium`, and `common`, none of which are cardinal directions. Once you are used to this distinction, you will find MOOs are just as easy to navigate. You will need to pay close attention to find the exits. To move to the Curtis Common, for example, you can use `go common` (if you were in the room with the exits shown previously) or just `common`.

While basic navigation is fairly easy, MOOs also have advanced forms of navigation (and teleportation). The following commands are useful for wandering around a MOO. Remember, however, not all commands will work on all MOOs.

`whereis <player>` or `whereis #<player object number>` is useful for locating another player. This command shows the players object number and the name of the room (and its object number). Following is an example of the output from this command:

```
Andrei [GPC] (#10278) is in The E&L Garden (#11).
```

Once you find out where someone is, you then can use the `@go` command to teleport your character to them.

`@go <name of a room or location>` or `@go #<object number of a room or location>`

For example, if you want to join Andrei, whom you know is in the E&L Garden, you could type `@go e&l garden`. Because you know the object number, however, you also could type `@go #11`. If you want to join `Cyan_Guest` in the Library, and entered `@go library`, you would see the following:

```
@go library
"library" is ambiguous.
Matches: The Library, Library Lounge, Library Foyer, Library (room of
 Features), Library (room of Player Classes), Library (room of Rooms), and
 Library (room of Things)
```

To actually get there, type **`@go the library`**.

`@go` *`<name of a room or location>`* or `@go #`*`<object number of a room or location>`* teleports you to a specific location.

`@join` *`<player name>`* or `@join #`*`<player object number>`* teleports you to a specific player. If you don't know where someone is or don't feel like looking it up, just use `@join` to go right to that player.

`home` takes you to the room you have designated as *home*. As a default, this is the room you start out in when you log onto the MOO as a new character.

`@sethom` sets a new home for your character. You will need to get permission before you can designate someone else's room as your home.

`@room` displays list of rooms you have remembered. If there is a room you plan to visit often, you can assign it a nickname. This makes it easier to get to that room.

`@addroom` *`<nickname of room>`* `#`*`<object number of the room>`* saves rooms for you for later use. Once you know the object number of a room, you can assign it a nickname for easier travel. For example, if you `@addroom club #8676`, and then type `@go club`, you will end up in room `#8676` and will longer need to remember the object number.

`@rmroom` *`<nickname of room>`* removes unwanted rooms from your `@rooms` list.

`@gag` *`<player>`* adds *`<player>`* to your `gag` list. When a player is on your `gag` list, you no longer see any text originating from that player. You no longer will see `pages`, `says`, or `emotes` from that player. Using this command is a good way to silence annoying players who repeat obnoxious messages.

`@ungag` *`<player>`* removes a designated *`<player>`* from your `gag` list.

`@gaglist` gets a complete list of the players you are `gag`ging. This command also tells you who is `gag`ging you.

`@sweep` gives you a list of everything (players and objects) in your vicinity that potentially could be listening to your conversation. Use this before considering MUD sex or any other private conversation.

Special Features

MOOs have many special features and new commands that may not appear on all existing MOOs. The following sections discuss some of these new commands.

Proprietary Commands

These commands have been added to the core of the MOO on a specific MOO. These commands work for all players. Getting access to the commands requires no actions on the part of the user. The availability of these commands will vary widely among MOOs. This section is really just to acknowledge that they exist.

`walk to <location>` is taken from Jay's House MOO, and does not work on LambdaMOO. Instead of teleporting you directly to `<location>`, this command figures out a path and walks you to the location you specify, just as if you had navigated using the standard directional commands. Using this command enables you to stroll through the virtual scenery and see who is residing where.

Features

A *feature* is a command or set of commands that has been created by MOO programmers for your use. Unfortunately, while many of the features appear on many MOOs, they may take different forms.

The following is a sample feature:

```
sign <message>
```

This command does not work on Jay's House MOO, but does work on LambdaMOO. It displays your message to everyone in the room with you in the following form:

```
sign <message>

                                  _____________
                                 |             |
Your_name holds up a BIG sign:   |  <message>  |
                                 |_____________|
```

You will find, however, that as a new MOO character (or a guest), you cannot use the `sign` command. Instead, you will need to use the `@addfeature` command, as in the following:

```
@addfeature #<object number>
```

The `@addfeature` command adds a designated feature to your character. From now on, you will have this feature and the command or commands it has at your disposal.

If you get tired of having a particular feature, you can remove it using the following command:

```
@rmfeature #<object number>
```

The `@features` command displays a list of the features you currently have activated for your player.

The following code shows the `@features` command at work:

```
@features
Feature Name
------- ----
#11907  Conferencing FO
#4361   Stage-talk and pose feature
#12308  Rave's Super God-Like Features
#16688  DT's Ragin' and not so Ragin' FO
------- ----
4 features found.
```

There is one problem with `@features` command. In the preceding example, you can see feature `#4361 Stage-talk` and `pose feature`, which provides the `sign` command. To use the `sign` feature, you need to manually add it to your character using the `@addfeature` command. To do this, type

`@addfeature #4361`

To enable the sign command on LambdaMOO, you need to use the `@addfeature #5023` command. Most of the MOOs have the same *features*, but they all have different feature numbers. The preceding example is from Chiba Sprawl MOO where the `sign` command is part of feature #4361, while on Lambda MOO the `sign` command is part of feature #5023.

So the problem is that each existing MOO probably has a feature for the `sign` command, and is likely to have a different object number. The best way to find out the object number is to ask. Players are more than willing to give you object numbers to check out.

TIP

If a player gives you an object number and doesn't tell you what the features do, you can't figure them out, or you just forget, remember to use the `help` command.

`help #<object number>` gives you the run down (and the syntax) for any commands the feature may have just added to your character.

Emotions and Feelings

Earlier, in Chapter 5, feeling commands were discussed. `smile <name>`, for example, enables you to smile at a specific person and provides a prospective. Look at the following example:

```
smile jen
You smile at jen.
And jen sees:
tarod smiles at you.
Everyone else sees:
tarod smiles at jen.
```

The preceding example came from LambdaMOO (where names are case sensitive rather than always being capitalized), which shows that these types of feelings are available on LambdaMOO. To add them to your character you will need to type the following:

```
@addfeature #21132
@addfeature #40842
```

On Chiba Sprawl MOO, the features for similar commands are `#207` and `#257`. The commands, however, may be somewhat different so you will want to use the local `help` command.

COMMAND

Adding the features `#21132` and `#40842` on LambdaMOO enables you to use the following commands:

`blake` *<thing>*	You are simply ignoring *<thing>*.
`blush` *<thing>*	*<thing>* causes you to blush.
`bow` *<thing>*	You bow gracefully at *<thing>*.
`chuckle` *<thing>*	You chuckle politely at *<thing>*.
`cackle` *<thing>*	You cackle madly at *<thing>*.
`comfort` *<thing>*	You comfort *<thing>*.
`cringe` *<thing>*	You cringe away from *<thing>*.
`cry <thing>`	You cry at *<thing>*.
`eye` *<thing>*	You eye *<thing>* warily.
`eyeball` *<thing>*	You give *<thing>* the hairy eyeball.
`feh` *<thing>*	You feh at *<thing>*.
`french` *<thing>*	You embrace *<thing>* in a long, passionate kiss.
`giggle` *<thing>*	You giggle at *<thing>*.
`glare` *<thing>*	You glare at *<thing>*.
`grin` *<thing>*	You grin at *<thing>*.
`grump` *<thing>*	You grump at *<thing>*.
`hug` *<thing>*	You hug *<thing>*.
`ignore` *<thing>*	You studiously ignore *<thing>*.
`kiss` *<thing>*	You kiss *<thing>*.
`laugh` *<thing>*	*<thing>* causes you to fall down laughing.
`mess` *<thing>*	You mess up *<thing>*.
`nod` *<thing>*	You nod to *<thing>*.
`paperwork` *<thing>*	You make *<thing>* do paperwork.
`pat` *<thing>*	You pat *<thing>* on the head.
`poke` *<thing>*	You poke *<thing>*.

`rtfm <thing>`	You tell *<thing>* to RTFM.
`shrug <thing>`	You shrug noncommitally at *<thing>*.
`sigh <thing>`	You sigh loudly at *<thing>*.
`smile <thing>`	You smile at *<thing>*.
`smirk <thing>`	You smirk at *<thing>*.
`toy <thing>`	You toy idly with *<thing>*.
`waggle <thing>`	You waggle your finger sternly at *<thing>*.
`wake <thing>`	You wake up *<thing>*.
`wave <thing>`	You wave to *<thing>*.
`wink <thing>`	You wink to *<thing>*.
`yawn <thing>`	You yawn at *<thing>*.

<thing> is an object in the room with you, which often will be a player, but actually can be any object or series of objects. If the object you want to use has more than one word in its name, you will want to use quotes, such as `"the name of the thing"`. You also can use the following for multiple "things":

```
<thing1> and "the name of thing2" and <thing3>
```

Some commands may not support the inclusion of multiple `things`, but many will. You also can use some commands by themselves. `smile`, for example, will depict you smiling.

So you now can also have the great feelings that are available on LPMUDs and DikuMUDs from a MOO. I think you will find these are very useful for expressing yourself. And the fact that they do have a perspective (the target sees the action differently from everyone else, unlike mimicking them with the `emote` command) adds more depth to the interaction.

Summary

You now should have what it takes to be an active social MUDder. If you are interested in combat MUDs, move on to Chapter 8!

8

CHAPTER

LPMUDs: An Introduction to Combat MUDs

Combat MUDs are the MUDs that go beyond just the basic social framework that all MUDs have by adding an integrated game system. Usually this game system consists of computer-generated monsters and a built-in system through which players fight with and kill these monsters. By killing more monsters, the players can advance their character in levels, making the character more powerful. Of course, because there are levels and gold there are all sorts of new political situations that can evolve. And it often is to a player's benefit to team up with other players to advance more quickly. This chapter discusses the basics of combat MUDs and goes into the specifics of LPMUDs which are used to present the basics of combat MUDs. Chapter 9 will address DikuMUDs, which are the other popular form of combat MUDs.

Combat MUDs

Previous chapters have discussed the basics of combat MUDs and social MUDs. In the preceding chapter, you learned more about specific types of social MUDs. This chapter provides the same type of information about combat MUDs.

There are two primary MUD systems that are used to run combat MUDs—the LPMUD system and DikuMUD. These two types of MUDs are so different that a chapter has been devoted to each. Other combat MUD systems are becoming popular, but most are spin-offs of LPMUDs (such as MudOS) or DikuMUDs (such as Circle and MERC). DGD is another new system that often is used to emulate LPMUDs, and has also been used to emulate a MOO.

This chapter focuses primarily on the LPMUD system and its use as a combat MUD system. This chapter relies on much of what has already been discussed throughout the book, as many of the previous examples in the book were taken from an LPMUD. LPMUDs and DikuMUDs make up the largest part of the MUD population and definitely are worth checking out. Most are very sociable, so even if you don't play the game itself, it still is a fun place to hang out and talk.

Occasionally, players convert some of the more social MUDs into game-oriented MUDs. These MUDs border between a social and a combat MUD. They usually have some framework that allows for combat and advancement, but focus more on role-playing in a specific genre rather than the more hack-and-slash oriented combat MUDs.

Game-Oriented MUSHes and MOOs

This section covers a small group of MUSHes and MOOs that are similar to combat MUDs, but do not have the complete, sophisticated combat components built into the MUD itself. While not the norm, game-oriented MOOs and MUSHes can be found and use the same basic commands described in Chapter 7. These systems often allow you to build a character with experience points and stats, as discussed in the MUD Persona section of Chapter 3.

The best example of MUDs that fall into this category are a series of MUSHes based on a role-playing game called *Vampire: The Masquerade*, which uses the *Storyteller* role-playing system. There are over 10 different Storyteller MUSHes. Storyteller MUSHes are popular because of their atmosphere as a role-playing MUD, the popularity of the vampire/supernatural genre, and the real-world popularity of the Storyteller system. The Storyteller system is a commercial role-playing system by White Wolf Games Studio, which has given permission for the system to be used on these MUSHes.

Character Creation

The Storyteller system enables you to build characters with stats—strength, dexterity, stamina, manipulation, charisma, appearance, perception, intelligence, and wits. It also offers many skills, such as firearms, occult, investing, investigation, empathy, alertness, and brawl. And finally, there are supernatural powers possessed by those characters that are vampires, ghouls, werewolves, and mages.

New players create a character when they log in for the first time. Not only do they assign initial statistics and skills that they normally might generate, they also define their character's background, resources, and personality. This is the same system used in the Storyteller role-playing system. Once the character is generated, he or she is thrust into the world to explore and learn about the surroundings.

The Environment

Once in the world created on these Storyteller MUDs, there are many subplots and places to explore. The worlds on Storyteller systems are based on real cities, such as Pittsburgh and Albuquerque. Players go to clubs and interact with other players to try to find vampires and other supernatural creatures. The players may want to kill or join these supernatural creatures. Wizards on the system may introduce other subplots to the players.

Sometimes on these systems, wizards will define what a player's character does or does not know. So, many times, the player must role-play (through his character) his actions in the world as if he had no knowledge that there were supernatural creatures or activities taking place in the MUD world. The character somehow must learn about these supernatural creatures through the course of the game before trying to find them. Some of these MUDs pursue a very rigid structure in the way that they role-play, which some players do not like. Other Storyteller MUDs, however, are not so strict.

In the event of combat or some other action that might require an underlying system that is built in to the MUD, a *judge* is called. A judge is a staff member that has been empowered to arbitrate combat and other actions. The players describe their actions to the judge and then the judge describes the results and does whatever is necessary to create the effects of the combat and the character's actions.

The results of the character's actions often are decided by automatic dice rollers. Players roll the dice to decide there fate. The number of dice and the results needed are affected by the character's skills, stats, and the action that is being attempted. All these numbers are taken from the Storyteller role-playing game.

There are no monsters (in the automated, MUD sense) on these MUSHes; players interact only with each other. Storyteller MUSHes are very popular, often with 80 to 100 people logged on at any given time.

NOTE

The Dark Gift MUSH is a Storyteller MUSH set in Pittsburgh that you can check out (`128.2.21.47 6250`). If you decide to play on a Storyteller MUSH, you probably will need to buy the book, *Vampire: The Masquerade*, so that you understand the system.

Other Systems Similar to Combat MUDs

There are several commercial systems that resemble combat MUDs in the way that they work. The leader in this field is the *ImagiNation Network*, which is owned by AT & T.

ImagiNation Network enables you to install software that creates a graphical front-end to the MUD world. It works in a way that is somewhat similar to MUDs, substituting arrow keys for directions and changing the semantics of many commands (or adding a graphical interface to them). This makes using the ImagiNation Network quite different from using MUDs.

Even though it is beyond the scope of this book to teach you how to use the ImagiNation Network, the interface is nice, there usually are many players, and if you enjoy combat MUDs, it probably is worth exploring. The areas of interest for MUDders probably will be the fantasy games found in MedievaLand. The ImagiNation Network also has other multiplayer games, such as chess, blackjack, poker, and more. It is a nice setting for both socializing and gaming. If you enjoy MUDs, I recommend checking it out.

LPMUDs

Throughout this book, most of the examples have been taken from LPMUDs. As such, if you have gotten to this point, you probably are very familiar with how LPMUDs work. If you have skipped forward to this point and are not familiar with LPMUDs, I recommend that you read Chapter 3 on MUD personas. The description of stats and classes comes from RealmsMUD, an LPMUD that will be used as the benchmark for this section.

LPMUDs have a very powerful language that can be used to create worlds, guilds, and anything else one desires to create. Because this language is like the C programming language (which many people are familiar with), many people have rewritten or added to the basic LPMUD. So expect LPMUDs to be very diverse. See Chapter 13 for information on this programming language and some of the different versions of LPMUD.

Because many of the commands you will use on an LPMUD have already been explained (refer to Chapter 5), and because Chapter 3 on the MUD persona explained the stats, levels, and other characteristics from an LPMUD perspective, none of this will be repeated here. This section discusses other MUD components and commands that are LPMUD-specific and have not already been covered.

LPMUD Command Summary

For your convenience, Table 8.1 recaps many of the LPMUD commands that have been covered in earlier chapters.

Table 8.1. Basic LPMUD Commands.

Command	*Description*
Basic Commands	
`look` or `l`	Shows your surroundings.
`look at <object>`	Gives you detailed information about the object. This commonly is called the object's long description.

Command	*Description*
`inventory` or `I`	Gives you a list of the items you currently are carrying.
`score`	Outputs the vital characteristics that make up your MUD character. These statistics and numbers are the lifeblood of your MUD character and determine his or her capabilities in combat, spellcasting, and other MUD activities.
Communications Commands	
`say <message>` or `'<message>`	Broadcasts your message in the format `You say, <message>` to everyone in the same room with you.
`tell <player> <message>`	Relays your message to the designated player wherever on the MUD he or she may be. The designated player can be standing in the room with you or at the other end of the MUD. This command usually will cost your character spell points.
`shout <message>`	Broadcasts your message to everyone on the MUD. This command usually will cost your character spell points.
Object Manipulation Commands	
`give <object> to <player>`	Gives the object in your possession to a player in the same room with you.
`get <object>`	Gets an object from the room you are in and puts that object in your inventory (or your possession).
`drop <object>`	Drops an object you have in your possession. Once you have dropped the object, it will be in the room you currently are occupying.
Weapon and Armor Commands	
`wield <weapon>`	Enables you to wield a weapon that currently is in your possession. If you get into a fight, you will be using the weapon (and any extra power it gives you) rather than your hands.
`wear <armor>`	Enables you to wear armor that currently is in your possession. If you get into a fight, the armor will provide an added level of protection beyond what you normally might have.

A Virtual Tour

This section takes you on a virtual tour of the main city in RealmsMUD. As you move through the city, important sites are pointed out. The most obvious starting point is the church, because each time you log in that is where you start.

The Church

Using the *church* as a starting point is pretty standard on LPMUDs. The church also is important because if you die, this is where you will need to go to `pray` (this is discussed later in the "Death and Dying" section).

```
The Church

    You are in the main church of RealmsMUD.
    You see a set of stairs that go down to the healing waters of
    the Realms. There is a huge pit in the center, and a door in
    the west wall. There is a button beside the door.
    There is a clock on the wall.
    This church has the service of reviving ghosts. Dead
    people come to the church and pray.

******DON'T CAST SPELLS OR FIGHT IN THE CHURCH!*****

There are exits south, north, up, east, and down.
Zxaigon the utter novice (Mortal).
A magic portal, leading to many houses.
REALMS players rules
> s
You are in an open area south of the village church. To the east
is a substantial town. Forest blankets the hills to the west.
You can see the top of a massive board through the trees to the south.
    There are three obvious exits: west, east, and south.
Grudge i am finally back to positive exp. (Mortal).
Minstrel Rabidchild died and lost tons of xp, but is helping newbies anyway! Go
figure! (Mortal).
Xtreme the utter novice (Mortal).
A Short Dream Post.
Grudge leaves east.
> e
```

NOTE

`soc` is an abbreviation for *South Of Church.* This is a popular central meeting place for trading objects and gold.

Newbie Areas

```
A track going into the village. The track opens up to a road
to the east and ends with green lawn to the west. You notice
a small hole here.
    There are three obvious exits: north, west, and east.
Taishan is getting engaged sooner than you may think....:) (Mortal).
A small hole leading down—Newbie Area.
```

`A small hole leading down—Newbie Area.` indicates an entrance (`down` or `d`) to a *newbie* area. This and other newbie areas can be quite useful for the new player. They are the best place to go and earn some experience points and raise your character up a few levels.

```
> n
A small Iron gate stands before you. Can you open it? If so, come
to Newbieville and have a fun time! If not, maybe Talon's Keep to the
north of Flame would be a fun place for you to adventure.
     There is one obvious exit: south.
```

This is another newbie area—if you can open the gate, you can enter. (Note, however, that characters above a certain level— often tenth level— are not allowed into most newbie areas.) Then you can kill monsters and accumulate treasure to your heart's content.

Taverns and Pubs

```
> s
A track going into the village. The track opens up to a road
to the east and ends with green lawn to the west. You notice
a small hole here.
    There are three obvious exits: north, west, and east.
Taishan is getting engaged sooner than you may think....:) (Mortal).
A small hole leading down — Newbie Area.
Milo arrives.
> e
You are on the outskirts of the town. Short roads lead off to the south
and north. More shops can be seen to the east, and forest to the west.
    There are four obvious exits: north, south, west, and east.
> n
Booo arrives.
A small yard surrounded by houses.
    There are four obvious exits: north, south, east, and west.
> e
    You are in The Common Man's Tavern.
This is a venerable drinking establishment.
The tavern is the center of the RealmsMUD social circle.
The Party Booth is working. Just go up.


THE NEW TRUTH OR DARE ROOM IS RUNNING. Go down.
Alcoholic:
    Honey Mead               6 coins
    Guinness Stout          25 coins
    Whiskey                 50 coins
    Peach Brandy           180 coins
    Moonshine              630 coins
    Sembia Wine            880 coins
Non-Alcoholic:
    Cormyrian Spring Water  50 coins
    Moonshean Coffee       100 coins
    Apricot Nectar         280 coins
    There are three obvious exits: west, down, and up.
An obituary paper.
A top list of players less than level 30(smaller list).
A top list of the most experienced players(top list).
A user graph hanging on the wall.
A bulletin board containing 20 messages.
An Industrial sized Trash Bin.
A sundial.
> buy mead
You pay 6 coins for a mead.
That feels good.
> buy coffee
You pay 100 coins for a coffee.
This is coffee thick and rich as river mud.
```

Taverns and *pubs* also are a core part of many LPMUDs. They provide another central location and a good area to socialize. Also, buying drinks is a primary method of healing both hit points and spell points. So, after you have struggled with a dangerous monster, you will need to go the nearest tavern and have a few drinks to help your body heal. Then, you will need some coffee to sober yourself up (probably so you can drink some more). Some drinks may require high constitution or levels to be able to stomach them—but anyone can drink mead and water.

You can see the `obituary paper` using the command `read paper`. Using this command displays a list of the last 10 or 15 characters that have died on the MUD, along with what killed them. This sometimes is useful because, by seeing which monsters killed which characters (and what level they were), you can find out who the really nasty monsters are and you can avoid them.

You will see the `trash bin` in many public places on RealmsMUD. On other MUDs, it probably will take other forms, such as a big dragon, or its function may be integrated into the MUD as a whole. Use the command `trash <item>` to throw something in the trash. This destroys the item, which, in theory, makes the MUD run faster (providing that enough people throw away all their unwanted items).

Graphs and Sundials

```
> l at graph
User graph by Draconian of Genesis. Patches by Animal.
W= Wizard m= mortal. The time is now  1:07
Time                                  Users
   0000000001111111111222222222233333333334444444444555555555566666666667 users
   1234567890123456789012345678901234567890123456789012345678901234567890 Wizes
 0:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                     :49, 4
 1:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                         :45, 4
 2:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                            :42, 4
 3:WWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                                 :37, 2
 4:Wmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                                       :31, 1
 5:Wmmmmmmmmmmmmmmmmmmmmmmmmm                                      :25, 1
 6:Wmmmmmmmmmmmmmmmmmmmmm                                          :21, 1
 7:Wmmmmmmmmmmmmmmmmmmmmm                                          :21, 1
 8:Wmmmmmmmmmmmmmmmmmmmm                                           :20, 1
 9:Wmmmmmmmmmmmmmmmmmmmmm                                          :21, 1
10:Wmmmmmmmmmmmmmmmmmmmmmm                                         :22, 1
11:WWmmmmmmmmmmmmmmmmmmmmmmm                                       :25, 2
12:WWmmmmmmmmmmmmmmmmmmmmmmmmmmm                                   :29, 2
13:WWWmmmmmmmmmmmmmmmmmmmmmmmmmmmm                                 :31, 3
14:WWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                             :35, 3
15:WWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                        :40, 3
16:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                     :43, 4
17:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                 :47, 4
18:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                 :47, 4
19:WWWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                 :47, 5
20:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                  :46, 4
21:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                 :47, 4
22:WWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                :48, 4
23:WWWWWmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm                :49, 5
Average W:2 m:36 Total:38 (all times in EST)
> l at sundial
*** California  *** Fri Jan 27 20:58:47 1995.
*** New York    *** Fri Jan 27 23:58:47 1995.
*** Atlantic    *** Sat Jan 28 00:58:47 1995.
*** Maine       *** Fri Jan 27 23:58:47 1995.
*** Mountains   *** Fri Jan 27 21:58:47 1995.
*** Armidale    *** Sat Jan 28 15:58:47 1995.
*** Perth       *** Sat Jan 28 13:58:47 1995.
*** Greenwich   *** Sat Jan 28 04:58:47 1995.
```

The user *graph* and the *sundial* are both amusing, and sometimes useful, utilities that reside in the RealmsMUD main tavern. Many MUDs will have similar items in their main pubs. It is worth going into them just to see what is there. The sundial shows the times in areas that are local to RealmsMUD players.

Bulletin Boards

```
> l at board
You can set up new notes with the command 'note headline'.
Read a note with 'read num', and remove an old note with
'remove num'.
The bulletin board contains 15 notes:

1:      This is a great board(Rhinoceros, Sep 21)
2:      ideas for monster pose(Rhinoceros, Sep 21)
3:      the new Athos(Athos, Sep 23)
4:      Where to go in the poss downtime...(Someone, Oct 17)
5:      headline Anybody else got nscore probs?(Fiz, Oct 25)
6:      hehehehe(Someone, Nov 21)
7:      sundial(Erroneous, Nov 25)
8:      re:sundial(Someone, Dec  1)
```

```
9:      UKMudmeet(Aitch, Dec 6)
10:     Me(Mikie, Dec 8)
11:     re sundial(Someone, Dec 8)
12:     REboot again!(Mandorallen, Jan 4)
13:     re: Reboot again(Mikie, Jan 6)
14:     quests(Lacroix, Jan 12)
15:     alignment(Aerich, Jan 21)
```

Bulletin boards are plastered all over many MUDs, often covering virtually every topic. Unfortunately, they all work differently. In general, as you can see on this bulletin board, if you look at the board, it will tell you how to use it. For example, with this bulletin board you can use `read <number>` to read a particular message. You can post a new message on the bulletin board using the command `note <title of your message>` and then put in an editor so that you can compose your message.

Shops

Shops are very important on combat MUDs. The following is a standard LPMUD shop, and most will have a similar interface. Here you can buy items using the `buy` command. For example, if I `buy helmet`, I will get the Mithril Helmet (medium), but if I want the Steel Helmet (any), I would need to `buy helment 2`. To see all the helmets the store has available (thus making it easier to assign a number to one), type the command **`list helmet`**. You also can type `sell <item>` to sell an item (any object you are currently carrying) to the store. If you are curious about how much the store will pay you for your item, you can `value <item>` and the store will give you an estimate of what it will pay you. If you sell the item, however, expect to get no more than 1000 to 1500 gold from the shop—even if the estimate you got was 10,000 gold. Most shops are limited to 1000 gold to keep players from making too much money. Sometimes your character's charisma stat may affect the amount of money the store is willing to give you.

```
> w
A small yard surrounded by houses.
    There are four obvious exits: north, south, east, and west.
> s
You are on the outskirts of the town. Short roads lead off to the south
and north. More shops can be seen to the east, and forest to the west.
    There are four obvious exits: north, south, west, and east.
> e
A long road going through the village. There is a hole leading down.
The road continues to the west. To the north is the shop, and to the
south is the adventurers' guild. The road runs towards the shore to
the east.
    There are five obvious exits: north, south, east, west, and down.
> n
You have entered the General store of RealmsMUD.
Items that you find while adventuring may be bought and sold here.
Commands are:
```

```
'buy item', 'sell item', 'sell all', 'list', 'list weapons',
'list armors' and 'value item'.

** NEW ** commands are.......
(You can now list armors by type)
eg. list helmet, list cloak, list boots, list ring,
    list amulet, list shield, list armor, list gloves.

Behind a small sales counter, there is a hall leading north.
A sign up above the hall reads: MANAGEMENT ONLY
    There are four obvious exits: south, north, east, and west.
An Industrial sized Trash Bin.
> list
1000:   Mithril Helmet (medium).
3000:   Magical Teddy Bear (any).
1000:   Steel Helmet (any).
1000:   Ring of Protection +1 (medium).
1000:   Magical Bracers (any).
3000:   Finely Made Scalemail (medium).
4000:   Chain Mail (any).
2400:   Elven Cloak (medium).
 300:   An empty keg.
1400:   A Halberd.
 800:   Silver Amulet (any).
1400:   Guardsman sword.
2000:   Ruby Scimitar.
1000:   Ghost Chains.
 800:   A glowing ruby amulet (any).
1000:   An old short sword.
1400:   Glowing Amulet (any).
1400:   A shortsword.
 400:   A dark orb.
 200:   White Robes (any).
 200:   Angelic Mask (any).
  80:   A Bag of Holding.
1000:   Pearl Spear.
 374:   A emerald tiara.
 596:   A copper bauble.
1506:   A diamond and platinum tiara.
 200:   Heavy long sword.
 400:   A Multi-colored Scroll.
 750:   a crumbling vellum scroll.
 998:   A Ruby Ring (any).
> s
A long road going through the village. There is a hole leading down.
The road continues to the west. To the north is the shop, and to the
south is the adventurers' guild. The road runs towards the shore to
the east.
    There are five obvious exits: north, south, east, west, and down.
> s
                 Welcome to The Adventurer's Guild
You have to come here when you want to advance your level.
You can also buy points for a new level.
Commands:  cost, advance, spend, list (number).
There is an opening to the south, and some blue shimmering
light in the doorway.
```

```
    There are two obvious exits: north and south
a book in a chain.
```

Combat

The *combat* system used by LPMUDs is very straightforward. There are two important numbers—weapon class (WC) and armor class (AC)—in the LPMUD system. Players generally never see these numbers, but can guess at them. Sometimes you can also get an idea of what they are from spells. Mages often have a spell called identify that gives a good idea of a weapon's WC and a piece of armor's AC.

Engaging in Combat

Before going any further, combat varies widely from MUD to MUD, but there is one command that you will always find useful.

`kill <monster name>` or `kill <character name>` enables you to initiate combat with the monster or character specified. (Note that killing other player's characters is never very nice and is sometimes against MUD rules.) Unless otherwise specified, this combat will take into account only weapons that you have wielded and armor you have worn at the time of the combat (and any that you might add during combat). Using spells and other special capabilities that your guild or class might possess will require special, MUD-dependent commands.

The `kill` command is used almost universally by combat MUDs. Now that you know how to get into a combat, read on to find out how combat works.

Anatomy of a Monster

First, learn a little about your possible opponents. The following session shows you what a monster in a room looks like and what the monster looks like close up.

```
You seem to have wandered into the lair of a few slimy green mephits. They do
not appear pleased at the intrusion. They wave their swords menacingly.
.==.  There are two obvious exits: north and west.
A Green Mephit.
A Green Mephit.
> l at mephit
An impish little mephit stares at you curiously.
Mephit is unharmed.
```

```
        Mephit is carrying:
An old short sword (wielded).
>
```

In the preceding example, you can see that the Green Mephit is carrying and wielding an old short sword. Just like player characters, monsters can carry items and even wield weapons and wear armor. They can also carry gold.

A Sample Combat Session

Okay, now you want to kill the poor defenseless mephit. The mephit is a pretty weak creature that cannot really hurt you, but is a good sample combat session. (My character on this MUD is a wizard, and I have edited my hit points to represent what they might look like for a real player.)

```
> kill mephit
Hp: 99    Sp: 99
You punch Mephit in the face, sending it staggering backwards.
Mephit missed Tarod.
Hp: 99    Sp: 99
You beat Mephit into a bloody pulp, causing blood to fly everywhere.
Mephit grazes you with a quick jab.
Hp: 98    Sp: 99
You punch Mephit in the stomach, knocking the wind out of it.
Mephit connects with a feeble jab to your head.
Hp: 93    Sp: 99
Mephit died.
You killed Mephit.
> l at corpse
This is the dead body of Mephit.
        Corpse contains:
273 gold coins.
An old short sword.
> get all from corpse
273 gold coins: Ok.
An old short sword: Ok.
```

After I attack the mephit, the MUD gives me updates of my hit points and spell points after every round of combat (explained in the next few sections). You can see my hit points going down (although not much) from the mephit's attacks. After `killing` the mephit, I `look` at its corpse to see what the mephit possessed. (Dead things leave corpses on MUDs, just like in real life.) And finally, the all important `get all from corpse` enables me take the spoils of victory from the dead mephit's body.

TIP

Remember the `get all from corpse` command. Use it whenever you kill a monster. You don't want to leave items and gold that you have rightfully earned lying around the MUD.

Weapon Class (WC)

Weapon class (*WC*) comes from the weapon you wield. Usually it is a number between 1 and 20, although occasionally there are super-powered weapons that go higher. Your hands have a WC of 3. This number is tied to the weapon, so when you wield a new weapon, your WC will become that of a weapon plus any bonuses you may have. You might get bonuses to your WC from your level, your guild, or certain stats. This varies from MUD to MUD, but being a fighter and having a high strength and dexterity will be the most likely things to increase your WC. On some MUDs, WC comes only from weapons and is not increased by anything else.

Armor Class (AC)

Armor class (*AC*) is a little more complicated. Because you can wear more than one kind of armor, this number is the sum of all the armor you wear. For example, you may have a suit of plate mail armor that has an AC of 4, a shield which has an AC of 1, and a magic ring with an AC of 2. If you were wearing all of these, your AC would be 7. Usually, characters have a default AC of 0. Table 8.2 shows the different types of armor and their values.

Table 8.2. Different types of armor and their values.

Type of Armor	*AC*
Amulets	1-2
Body Armor (such as plate mail or chain mail)	1-5
Boots	1-2
Bracers	1-5
Chain Mail	1-4
Cloaks	1-2
Gauntlets or Gloves	1-2
Helmets	1-2
Leather	1-3
Plate Mail	4-5
Ring Mail and Scale Mail	1-4
Rings	1-2
Shields	1-3

Remember that these armor class numbers are guidelines. They can change at any time, and there are always special objects that fall outside the normal range. But these should help you at least guess what the AC of an item might be.

Usually there are guild powers and items that can be used to either identify an item (giving a rough idea of its WC or AC) or compare two items showing their relative power. You also can try two items (in this case, probably weapons) and see which one seems to work better; however, this usually isn't very reliable. (LPMUD combat does have random elements, so even if one weapon is better, you might just be experiencing some bad luck when you try it.)

When you fight, your WC is matched up against your opponent's AC, and vice versa. The larger the advantage you have in the WC to AC ratio, the more damage each round you will do to your opponent. The better AC to WC ratio that you have, the less damage you will take from your opponent.

Rounds and Damage

Two important parts of the combat system are *rounds* and *damage*. LPMUD combat is divided into rounds. Each round, you get an attack and your opponent gets an attack. Intermingled with rounds, you also can use any special capabilities you may have, such as spells, guild abilities, or items. On most MUDs, you can only use one of these items each round, but some MUDs have no restrictions. On MUDs with no restrictions, you can cast as many spells and use as many special capabilities and items in a round for which you can type the commands.

Damage also is a pretty straightforward concept. When you are hit, you will take damage. This damage shows up as a deduction from your hit points. The more damage you take, the lower your hit points will go. If these points go to zero, you die. To keep from dying in combat situations, you will want to use `wimpy`, which is a very useful command.

`wimpy <number>` allows you to set your character's *wimpy*. Wimpy is a number between 1 and 100 that represents a percentage of your character's total hit points. If your current hit points fall below that set number, you will automatically run away from a fight. If you never want to run away from a fight, set your wimpy at `0`. Suppose that when your character is fully healed, he or she has 200 hit points and his or her wimpy is set to 50, and then he or she goes into combat. If your character is dealt a blow that pushes his or her hit points below 100, your character will run away. Unfortunately, he or she will run in a random direction through the available exits.

Some MUDs have the additional command `wimpydir <direction>` that enables you to set the default direction your character will take when he or she wimpies.

Set your wimpy at `50`. Better safe than dead.

Remember to heal yourself regularly (especially after you wimpy). As shown in the Virtual Tour, you can heal in pubs. You also can find items to heal yourself—look for healing potions and carry them with you. Some guilds also can heal (usually priests or clerics). You may have to run back and forth between a monster and a pub several times before you can kill the monster.

Partying

This kind of partying is not the traditional drunken revelry. MUD *partying* is forming a party of adventurers that can kill monsters together. Some MUDs give characters the capability to band together and share experience as they kill monsters. On many older MUDs, only the player that actually deals the killing blow to a monster receives any significant experience points. Forming a party allows the computer to automatically divide among the party members all the experience gained. Sometimes the experience is divided equally, sometimes the members of the party divide the number of shares, and sometimes the computer decides based on the levels of the party members.

The specific commands for forming parties and sharing experience varies widely. There is no standard format for partying. You will need to ask around on the MUDs that you choose to play to get the specifics for that MUD.

Death and Dying

Following is the most dreaded thing you will ever see on an LPMUD:

```
You die.
You have a strange feeling.
You can see your own dead body from above.
```

If it happens, you will need to find your way back to the church and do the following:

```
> pray
You feel a very strong force.
You are sucked away...
You reappear in a more solid form.
```

Unfortunately, praying has its penalties. On RealmsMUD, you will lose one-third of your experience points when you die. You also will lose two points in your stats—usually one point each from two random stats (such as strength, intelligence, dexterity, wisdom, constitution, or charisma).

Advancement

When you use the `score` command, you will see a lot of information about your character, but the most important line will be

```
You have 22% of the experience needed for the next level.
```

or the number of experience points you have. If you see the preceding line with a percent sign, and it goes over `100%`, it is time for you to advance. If the MUD you are on provides you with your actual number of experience points (rather than the percentage), you will need to use `help levels` or `<guildname> levels` to find out how much experience you need for your next level. Once you pass the listed number of experience points for your next level, it is time to advance.

When it's time to advance, you will need to go to your guild and advance. The following is an example of what you might see in a guild room (this example is the fighters' guild):

```
Everyone one of us has heard the call! Brothers of true metal, proud and
 standing tall! We know the power within us has brought us to this hall!
 There's magic in the metal! There's magic in us all!

 You now stand in the middle of the main hall in the Fighter's Guild.
 The ancient walls of stone are covered with many tapestries of past and
 present heroes. Perhaps your face will appear here one day!

 The images of Animal, Gor, and Barbie appear here dressed in
 full battle armor and wielding their own fearsome weapons of steel!

Here you may: join, advance, points, spend, fix, and list.
New command: choose <subclass>. you must pick barbarian or paladin!


      There are three obvious exits: south, west, and north.
>
```

`join` enables you to join the guild. You must be an adventurer or have no guild before you can join. There are special places where you can renounce your guild (which has a significant experience point penalty) so that you may join a different guild.

`advance` enables you to advance a level after you have accumulated enough experience points (you must go through your guild to do this). Advancing probably will give you new skills and certainly will improve your hit points and spell points. You also will get one stat point to spend on increasing you stats.

`points` shows you how many stat points you currently have to spend.

`spend <stat>` enables you to spend a stat point. When you use the `spend` command, it increases the designated stat by one. `<stat>` can be `str`, `dex`, `int`, `con`, `wis`, or `chr`, for the respective stat.

continues

`fix` enables you to fix the occasional weird things that happen on MUDs. For example, sometimes things break or fail to load properly. If your guild commands suddenly do not work, go to your guild and use the `fix` command.

`list` enables you to see which quests you need to complete. (For more information about quests, see Chapter 11.)

Summary

This chapter is just the beginning of what you can do on LPMUDs—there is much more to learn! Because each LPMUD has its own idiosyncrasies, enhancements, and modifications, they may act and behave differently than what has been described in this chapter. You will find that many MUD players are friendly and helpful with newbies. Often, other players will give weapons, armor, and gold to new players. Sometimes there are strings attached and sometimes it is just for fun. If you have any questions, ask; someone probably will answer you.

You'll find that once you get the basics, it is very easy to learn about all the new things you will encounter in the MUD world. Don't forget, there are always people to ask when you need help. So get online and check it out.

9

CHAPTER

DikuMUDs

Since the first MUDs appeared in the 1970s, their development has gone in one of two directions: combat and social. The combat games (often called “hack-and-slash”) traditionally center on the violent exploits of each individual character, whereas social MUDs lack the “gaming” element of MUDding. DikuMUDs, named after the Danish abbreviation for the University of Copenhagen where it was developed, are a compromise between these two basic styles. Adventuring and combat are the mainstays of the Diku world, but players usually are encouraged by the types of situations and problems which they face to work together and communicate.

Thus, the DikuMUD is a world of interactive gaming that stresses player collaboration and cooperation along with the sheer combat activity of a player's individual “character.” Many of the new DikuMUD systems offer elaborate “clan” systems that bring a political element into the game. Combined with CPU-efficient coding for fast operation and support for numerous simultaneous users, a large “virtual world,” and enhancements that increase user-friendliness and playability, the DikuMUD community is one of the fastest growing online game platforms on the Internet.

Finding DikuMUDs

The Internet newsgroup `rec.games.mud.announce` is a good place to look for new DikuMUDs—postings to this group announce the status of new MUDs of all types. For information specific to Diku systems, the newsgroup `rec.games.mud.diku` is dedicated to the discussion of DikuMUD gameplay, programming, and related topics. Also, the DikuMUD FAQ (list of frequently asked questions about DikuMUDs) often is posted here.

Tour of the DikuMUD World

We now will take a tour of the DikuMUD world, exploring the "town of Midgaard"—a common feature to virtually every DikuMUD. Some of the most important functions of the Diku system will be illustrated and explained here, while more detailed descriptions of different gaming aspects follow later in the chapter.

After logging on to a DikuMUD system and creating a character—here we have connected to Realms of Magic (Internet address `p106.informatik.uni-bremen.de` or `134.102.216.8 4000`) and created a warrior character named "Bub"—the Diku system starts the new player off in the Temple of Midgaard.

```
> who
Players
------
 Doran the humble
 Bub the Swordpupil
 Shayla the Dalai Lamia
 Hendrek the Troll-Swordman
 Bobby the Fencer
 Tommyknocker the Recruit
 DeLormar the Levite
 Io the Warrior

8 characters displayed.
```

`who` prints a list of other players currently connected to the DikuMUD. Many systems will automatically paginate long lists of players—you will be asked to press <return> to view subsequent pages of text. A related command is `finger <playername>`, which returns information about a particular character on most Diku systems, often telling you if the player is logged on or when the player was last logged on.

The DikuMUD World

When you play a DikuMUD, you are submerged in a fantasy world that simulates several aspects of "real life" as a part of game interaction. You must guide your character around the world, command him or her to pick up and wear objects, and even occasionally eat or sleep. The command interface is relatively simple, relying on command words and simple sentences to order your character to perform certain actions. The following sections describe the commands necessary to perform basic actions such as walking, manipulating objects, and wearing equipment.

Movement

The first concern of the new DikuMUD player is movement. How is the world navigated? How is the environment described, and what parts of it can be manipulated? This first part of the tour presents these MUD basics, readying the new player for exploration of new "virtual territory."

Direction Commands

Maneuvering your character around the DikuMUD is much the same as on the MUD. You "walk" by typing compass directions: `north`, `south`, and so on. You can abbreviate these commands as `n`, `s`, `e`, `w`, `u` (for up), `d` (for down), and so on.

NOTE If you have the speedwalk function engaged, you may have difficulties maneuvering if your DikuMUD contains diagonal directions (such as NW for northwest). See "Navigating the DikuMUD World" for more information.

Room Descriptions

The descriptions of the DikuMUD's *virtual rooms* contain information about the appearance of the room, the name of the room, objects contained in the room (including monsters or other players), and the visible exits from the room. Notice all of the objects, room exits, and area descriptions in the following session:

```
> w
The Temple Of Midgaard
  You are in the southern end of the temple hall in the Temple of Midgaard.
The temple has been constructed from giant marble blocks, eternal in appear-
ance, and most of the walls are covered by ancient wall paintings picturing
Gods, Giants, and peasants.
  Large steps lead down through the grand temple gate, descending the huge
mound upon which the temple is built and end on the temple square below. To the
west, you see the Reading Room. The donation room is in a small alcove to your
east.
The TOP TEN board is standing here.
An automatic teller machine has been installed in the wall here.

> s
The Temple Square
  You are standing on the temple square. Huge marble steps lead up to the
temple gate. The entrance to the Clerics Guild is to the west, and the old
Grunting Boar Inn is to the east. Just south of here you see the market square,
the center of Midgaard.
A large fountain carved from blue-streaked marble is here, bubbling merrily.
A Peacekeeper is standing here, ready to jump in at the first sign of trouble.

> s
```

```
Market Square

You are standing on the market square, the famous Square of Midgaard. A large,
peculiar looking statue is standing in the middle of the square. Roads lead in
every direction, north to the temple square, south to the common square, east,
and westbound is the main street.
> e
The Main Street

You are on the main street crossing through town. To the north is the general
store, and the main street continues east. To the west you see and hear the
market place, to the south a small door leads into the Pet Shop.
HonkMan the Recruit is standing here.
```

Movement Messages

The `you cannot go that way...` message indicates that you tried to go in a direction that does not have an exit. Note that some DikuMUDs include one-way doors, and areas in which passages change their routing (mazes) where you can easily get lost! Notice what happens when you attempt to move in an "illegal" direction:

```
> e
Alley at Levee
  You are standing in the alley which continues east and west. South of here
you see the levee.
A mercenary is waiting for a job here.

> n
Alas, you cannot go that way...
```

Short Description Mode

The descriptions provided by the DikuMUD contain a great deal of information about the environment. Note that you can abbreviate the display on most DikuMUDs by typing **`brief`**, which causes the MUD to send only the name of the room and objects and monsters/characters contained in the room—the description will not be displayed automatically. To switch back to `verbose` mode, where all information is displayed, simply type **`brief`** a second time.

Room descriptions usually appear as follows when `brief` mode is set:

```
>brief
Brief mode on.

> e
The Main Street
```

```
> s
Entrance Hall to the Guild of Swordsmen

> e
The Bar of Swordsmen
A waiter is here who looks like he could easily kill you while still carrying
quite a few firebreathers.
```

Equipping the Character

The capability to maneuver around the Diku world is not enough for success in the game, however. Characters must be properly equipped to deal with the rigors of coming battles. This section highlights essential aspects of play, from picking up objects and purchasing supplies to readying armor before a fight.

Getting and Carrying Objects

You will need equipment as you adventure in the DikuMUD world—the better armor and weapons you possess, the more able you will be to fight the aggressive denizens of the virtual world. `get <object>` will tell your character to pick up a specific item, and `get all` will cause the character to attempt picking up all "loose" objects in the room—up to the maximum carrying capacity of the character. Note that you have limited strength and "space" for objects in your inventory, and the stronger your character, the more you can carry (see "Character Attributes"). To see what you are carrying, simply type **`i`** for inventory.

```
Midgaard Donation Room
  You are in a small, undecorated room just off of the main temple. There are a
couple of small wooden benches here where people occasionally sit while they
wait for items to appear. The temple is to the west.
A piece of cloth is lying on the ground.
A buckler is lying on the ground.
A small used torch.
A small loaf of bread lies here, looking a bit green.
A small wooden sword with a short, stubby blade is here.

> i
You are carrying:
a small bread
an almost burned down torch

> get all
You get a cloth armor.
You get a buckler.
You get an almost burned down torch.
You get a small bread.
You get a small wooden sword.
```

Wearing Items—Equipping

Before your armor and other equipment will aid your character, you must `wear <item>` or `wear all` to attempt to wear all "wearable" items currently in your inventory. You also may `remove <item>` or `remove all` in the same manner to quit using a certain piece of equipment (this is necessary when you want to change from one piece of armor to another, for example).

```
> wear all
You start to use a buckler as a shield.

You wear a cloth armor on your body.
```

Wielding a Weapon

Before you fight, it is necessary to use the `wield <weapon>` command to prepare a specific weapon for combat, which essentially is holding a weapon in your hand (note that this function is not accomplished by `wear all`). As with the `wear` command, you may `remove <weapon>` to unwield it. In addition, you can `grab <weapon>` to hold a second weapon. Certain classes may be able to attack with *both* weapons, and on certain Diku systems, your character gets the hit and damage bonuses for the second weapon applied to the attack (see section on "Combat Modifiers" for detailed information).

```
> wield sword
You wield a small wooden sword.
```

List of Equipment

The equipment you currently are `wear`ing or `wield`ing does not appear in your inventory. Rather, it shows up in the `eq` list, which is displayed when you type `eq`. Items that are worn usually do not count toward the limited number of objects that you can hold in your inventory, but do add to the total weight that your character can carry.

The following text represents the `eq` command output on a "typical" DikuMUD:

```
> eq
You are using:
<worn on body>    a cloth armor
<worn as shield>   a buckler
<wielded>      a small wooden sword
```

Buying Combat Equipment

To be successful in combat, characters must be properly equipped. Stores cater to this need—provided you have the money to buy their goods. Weapons and armor are sold in shops on every DikuMUD:

```
> s
The Armory
  The armory with all kinds of armors on the walls and in the window. You see
helmets, shields and chain mails. To the north is the main street.
On the wall is a small note.
An Armorer stands here displaying his shiny new (and previously owned) armors.

> list
You can buy:
A buckler for 2 gold coins.
A pair of leather gloves for 75 gold coins.
A chain mail shirt for 2500 gold coins.
A pair of leather pants for 150 gold coins.
A leather cap for 150 gold coins.
A studded leather jacket for 500 gold coins.
A shield for 100 gold coins.
A breast plate for 18000 gold coins.
```

NOTE Stores like the one mentioned in the preceding code are scattered throughout DikuMUD worlds, offering to sell certain standard equipment, as well as special items that have been found and sold by other players.

Miscellaneous Shops

You will find stores and shops in Diku worlds that sell items other than armor and weapons. Merchants will sell you bags, torches, and other miscellaneous necessities. Also common are food and water sellers.

```
> e
Eastern end of Alley
  You are standing at the eastern end of the alley, the city wall is just east,
blocking any further movement. There is colorful graffiti sprayed on the wall.
A small warehouse is directly south of here and a mexican food
stand has been set up north of here.
A mercenary is waiting for a job here.
A mercenary is waiting for a job here.

> n
Uncle Juan's Eatery
```

```
  This place makes your nose run just from the smell of the spicy food they
sell. Spicy as it may be, it still looks very tasty. A box of taco shells is
set on the counter.
  There is a large, friendly sign hanging on the wall here.
Uncle Juan is here ready to take your order.

> list
You can buy:
Some nachos with cheese for 5 gold coins.
A spicy hot burrito for 10 gold coins.
A Mexican taco for 15 gold coins.

(later in the session…)
>
Ye Olde Water Shoppe
  You are in Wally's World O' Water, whose proprietor, Wally, is sure to have
something to meet all of your water needs. Wally the Watermaster is standing
behind the counter, proudly displaying his fine collection of contemporary
waters.

> list
You can buy:
A canteen of clear water for 45 gold coins.
A bottle of clear water for 10 gold coins.
A cup of clear water for 2 gold coins.

> buy canteen
Wally the Watermaster tells you "Sorry, but here, no money means no water!"
```

TIP

It is easy to forget to carry water with you as you adventure—be sure to always have a canteen, and type **`fill canteen fountain`** when in the temple square to replenish the supply periodically.

Store Commands

When you are in a store, you can `list` the available merchandise, `buy <item>` to buy something, `sell <item>` from your inventory, or `value <item>` to find out how much one of your items is worth. For example, if you try to buy food with no money, you will see:

```
> buy taco
Uncle Juan tells you 'No dinero! Hit the road!'
```

Donating Equipment

Most Diku systems support item *donation*, where items that you specify by typing `donate <item>` will be transported to a *donation room* somewhere on the MUD (usually a room east of the temple of Midgaard) where other characters may pick up and use your unwanted items. Make the following procedure a habit, as your donations will aid other players.

```
> i
You are carrying:
a small bread
an almost burned down torch

> donate torch
You donate an almost burned down torch. It vanishes in a puff of smoke!
```

Emotions

The Diku world is often quite descriptive. Many commands allow you to describe "everyday" actions and gestures. Showing "emotion" and mood in this way helps add to a character's personality:

```
> sigh
You sigh.

> wave juan
You wave goodbye to Uncle Juan.
```

TIP

To make your interactions with other players more interesting and "colorful," most DikuMUDs provide a number of commands that describe your emotional state, actions, and so on. `Wave` and `sigh` are just two of the commonly supported commands—see your individual DikuMUD's help screen for more information.

Training the Character

As essential as outfitting a character with weapons and armor is making sure that the adventurer is properly trained. Skills and spells allow characters to effectively fight, successfully cast their incantations, and better tackle the problems they face in the DikuMUD environment.

Guildmasters and Practicing

Characters in the DikuMUD world have *skills* (and sometimes *spells*) that must be trained before they can be used by the character. On most Diku systems, you cannot use untrained skills and spells at all—you will have a zero-percent chance of success. The higher the level your character obtains, the more useful and powerful skills and spells available to that character will be. Characters earn *training sessions* (listed in the score display) as they gain levels—these are in turn used to improve individual skills and spell capabilities. You can obtain a list of skills using the `sk` command (spells are listed with `spells` or `spe`). When in the presence of your guildmaster, use `pra <skill>` to improve a skill or spell in a training session. That is, a training session is subtracted from your total training sessions (listed under the *score* command display), and the proficiency of the character in the trained area is increased.

The following session demonstrates the output of the `skill` command (example uses the `sk` abbreviation), and a sample interaction with a guildmaster:

```
> sk
You know the following skills:

 Level              Skill       How Good
------------------------------------------------------------
  1                 kick       (bad)
  1                 bandage    (bad)
  1                 riding     (not learned)

> s
The Tournament and Practice Yard
Your guildmaster is standing here.

> pra
The Guildmaster says "This is what I can teach you:"

  1                 kick
  1                 bandage
  1                 riding
```

Skill Training Messages

When training skills or spells, the message `You Practice for a while...` indicates that you have successfully practiced a particular skill, while the message `You do not seem to be able to practice now` indicates that you have no remaining training sessions. The `You have now mastered that skill` message alerts you to the fact that you are completely proficient in a certain skill area or spell.

You will be informed by the MUD when you have successfully trained in a skill or spell. The Guildmaster will also tell you if you have insufficient practice sessions to train, usually in the following manner:

```
> pra kick
You Practice for a while...

> pra kick
The Guildmaster says "You do not seem to be able to practice now."
```

Character Health

Monitoring and maintaining the health of your character becomes important as you roam the online world, traveling great distances and fighting along the way. Food, drink, rest, and sleep are all important in keeping your character in peak condition. This section highlights the daily—yet important—process of eating and sleeping.

Score

The score command is universal on DikuMUDs—every Diku system will display the basic traits and status of your character in this manner, and many will give you even more detailed information including statistics, armor class values, and active magic spells or modifications that somehow affects the character. The section on "Character Informational Commands" provides a more detailed discussion of these individual attributes.

```
> sc
You are a 17 year old male Human and about 1.6 m tall.
 * It's your birthday today *
You have 24(24) hit, 100(100) mana and 80(83) movement points.
You are almost naked, and your alignment is 'Neutral'.
You have scored 1 exp, and have 0 gold coins.
You still have practice sessions left.
You need 1999 exp to reach your next level.
You have been playing for 0 days and 0 hours.
This ranks you as Bub the Swordpupil (Level 1).
You are standing.
```

Saving Your Game

Just like other work on a computer, your progress in a DikuMUD game must be saved to disk. While not having to do strictly with character health, the habit of saving your progress after major accomplishments may well save your (the player's) sanity! This saving process is normally done automatically, but you can force the system to save easily at any time:

```
> save
Saving Bub.
```

save instructs the DikuMUD to record the status of your character in its player files, so that there is a record of your current achievement in the case of a system failure or other event that would otherwise erase the changes to your character since your last save. Many systems automatically save every few minutes, but be sure to manually save after gaining levels, finding good equipment, and winning tough battles.

Food and Drink

In the DikuMUD world, you must eat and drink periodically to keep yourself from starving or dehydrating. Although you cannot die from either condition in the game, you will not regenerate (heal yourself and gain back spell and movement points) without doing both on a regular basis. The message `You are too full to eat more!` indicates that you are fully satisfied, and will regenerate normally.

It is wise to carry food in your inventory, so that you can eat when your character becomes hungry. The MUD will inform you when your stomach is filled.

```
> i
You are carrying:
a small bread
an almost burned down torch

> eat bread
You are too full to eat more!
```

Character Fatigue

Adventuring in the Diku world can be hard work, and your character will eventually need rest, to heal wounds quickly, and regain the ability to move around the MUD. Your character inevitably will suffer damage in combat, lose spell points, and run out of movement as you adventure in the DikuMUD world. You may periodically want to type `sleep` or `rest` to recover points faster. Note, however, that you cannot perform other actions while resting, and when sleeping, you cannot even see what is going on around you. Regeneration, however, is *much* faster when asleep. You then can `wake` your character, and `stand` to get back on your feet.

This process is accomplished easily, as follows:

```
> sleep
You go to sleep.

> wake
You wake, and sit up.

> stand
You stand up.
```

Combat

While social interaction forms bonds between Diku players and exploring provides hours of entertainment, the game still centers around combat. Through successful combat characters gain experience so that they can advance to higher levels, and obtain equipment and gold. This section provides an overview of DikuMUD combat, from initiation of battle to the final resolution of conflict.

"Considering" an Opponent

It often is useful to have a rough idea of the capabilities of an enemy before beginning combat. The `con <target>` command gives you a rough estimate of the difficulty of a battle with the target monster or individual. Note, however, that this estimate often does not take into account spellcasting capabilities and the like, which may make combat much more difficult—proceed into combat with care. Results of `You ARE mad!` and `You are a dumb player for even considering`, two messages common to many DikuMUDs, indicate that the target creature is *much* more capable than your character—beware! Notice the message in the following session:

```
> w
Main Street
A beastly fido is mucking through the garbage looking for food here.

> brief
Brief mode off.

> w
Main Street
You are at the end of the main street of Midgaard. South of here is the
entrance to the Guild of Magic Users. The street continues east towards the
market square. The magic shop is to the north and to the west is the city
gate.

> s
 Entrance to Mage's Guild
  The entrance hall is a small, poor lighted room.
A sorcerer is guarding the entrance.

> con sorcerer
You ARE mad!
```

The `con <target>` command (short for `consider`) gives you an estimate of the difficulty of a battle with the target monster or individual.

Initiating Combat

Your existence in the DikuMUD world revolves around combat. You often will have to begin a fight with one of the denizens of the MUD world, as only a few are aggressive enough to initiate combat themselves. The command `kill <target>` engages the target creature in combat. Note that some classes of character may begin combat in other ways (by casting spells, using the `backstab` capability, and so on)—these alternate methods are covered in the "Character Classes" section. Creatures known as *aggressive monsters* will attack when you enter the room, so caution must always be used when investigating new areas.

As soon as a player begins adventuring in the Diku world, they begin to engage in combat, as in the following sample session:

```
> e
Main Street
  You are on the main street passing through the City of Midgaard. South of
here is the entrance to the Armory, and the bakery is to the north. East of
here is the market square.
A beastly fido is mucking through the garbage looking for food here.

> con fido
Fairly easy.
> kill fido
You miss the beastly fido with your pierce.
```

`kill <target>` initiates combat against the target creature.

Issuing Commands in Combat

You can issue certain commands in combat, often initiating a skill attack possessed by the character (the `kick` capability is the example given here). Also, changes may be made to a character's worn equipment, wielded weapon, and used objects in the midst of combat. Furthermore, it often is possible to use items such as potions, scrolls, and other magic items during combat.

Combat messages indicate the success of the player (and his or her opponent) in dealing damage, as in the following session:

```
>
The beastly fido tickles you as he hits you.
You miss the beastly fido with your pierce.

> kick
Your beautiful full-circle kick misses the beastly fido by a mile.
>
The beastly fido misses you with his hit.
You miss the beastly fido with your pierce.
```

```
> kick
The beastly fido misses you with his hit.
You miss the beastly fido with your pierce.

> kick

The beastly fido misses you with his hit.
You miss the beastly fido with your pierce.

>
You miss your kick at the beastly fido's groin, much to his relief...
```

Skill/Ability Success or Failure

The DikuMUD will tell you whether you have been successful in a skilled action attack by printing a message to the screen (often in a humorous format!) that indicates the result. Note that many actions are "limited," meaning that they may only be performed once per a certain set number of combat "rounds."

Conclusion of Combat

Combat concludes with the demise (or flight) of one combatant. After combat, it is standard practice to search the corpse of an opponent for valuable equipment, gold, and weapons. The next session excerpt represents the end of a typical combat:

TIP

After combat, always remember to get coins and items from the corpse of your opponent.

```
>
The beastly fido misses you with his hit.
You barely pierce the beastly fido.
The beastly fido is stunned, but will probably regain consciousness again.
The beastly fido is mortally wounded, and will die soon, if not aided.

>
You barely pierce the beastly fido.
The beastly fido is dead! R.I.P.
You receive 36 experience points.
Your blood freezes as you hear the beastly fido's death cry.

> get all corpse
The corpse of the beastly fido seems to be empty.
```

NOTE

When typing commands that refer to multiple objects, you may leave out the words *in* and *from*. For example, you may type **`get all corpse`** rather than `get all from corpse`, although both commands will produce the same result. Similarly, `put bread bag` accomplishes the same thing as `put bread in bag`.

Fleeing Combat

Sometimes you will become involved in a combat that proves too difficult for your character. In order to avoid death, you can flee from battle and fight another day, as in the following session:

```
> s
Eastern end of Alley
  You are standing at the eastern end of the alley, the city wall is just east,
blocking any further movement. There is a colorful graffiti sprayed on the
wall. A small warehouse is directly south of here and a mexican food
stand has been set up north of here.
A mercenary is waiting for a job here.

> con mercenary
Do you feel lucky, punk?

> kill mercenary
You miss the mercenary with your pierce.

> kick

The mercenary pierces you.
That Really did HURT!
You miss the mercenary with your pierce.

> You miss your kick at the mercenary's groin, much to his relief...

The mercenary pierces you.
That Really did HURT!
You miss the mercenary with your pierce.

>
The mercenary misses you with his pierce.
You miss the mercenary with your pierce.

> flee
Uncle Juan's Eatery
  This place makes your nose run just from the smell of the spicy food they
sell. Spicy as it may be it still looks very tasty. A box of taco shells is set
on the counter.
  There is a large, friendly sign hanging on the wall here.
Uncle Juan is here ready to take your order.
You flee head over heels.
```

The *wimpy* Command

It is a good idea to be prepared for the worst in combat. If a foe is too tough, you may not have time to manually flee, so Diku systems allow you to preset a point at which your character will run away with the `wimpy` command. Death is always a risk, however, and nearly every player will meet his or her demise at least once in the course of play:

`wimpy` tells the DikuMUD that you want your character to "run away" if you fall below a certain amount of hit points. `wimpy` *<number>* sets this amount, and `wimpy 0` forces you to stay in battle to the bitter end—you will never run away (be cautious with this setting!). A PANIC message means that you failed to escape from your foe, and must continue combat.

```
> l

Ye Olde Water Shoppe
A mercenary is waiting for a job here.

> kill mercenary
You miss the mercenary with your hit.

>
The mercenary pierces you.
That Really did HURT!
You miss the mercenary with your pierce.

>
The mercenary misses you with his pierce.
You miss the mercenary with your pierce.

> wimpy 5
The mercenary barely pierces you.
You miss the mercenary with your pierce.

>
OK, you'll wimp out if you drop below 5 hit points.

>
The mercenary misses you with his pierce.
You miss the mercenary with your pierce.

>
The mercenary barely pierces you.
You wimp out, and attempt to flee!
PANIC! You could not escape!
```

Death

Dying in most DikuMUDs is not the end of the world, or the game. Consequences of death vary from MUD to MUD, but commonly you lose experience and often your equipment. Most MUDs, however, enable you to retrieve your corpse and your equipment—if you can get to it! Other players may be able to help you find your "remains" in these circumstances.

```
>
The mercenary pierces you.
You're stunned, but will probably regain consciousness again.
All you can do right now is think about the stars!

>
The mercenary pierces you hard.
You are mortally wounded, and will die soon, if not aided.

> sc
You are a 17 year old male Human and about 1.6 m tall.
 * It's your birthday today *
You have -9(24) hit, 100(100) mana and 79(83) movement points.
You are almost naked, and your alignment is 'Neutral.'
You have scored 34 exp, and have 0 gold coins.
You need 1966 exp to reach your next level.
You have been playing for 0 days and 0 hours.
This ranks you as Bub the Swordpupil (Level 1).
You are mortally wounded! You should seek help!
>
The sun slowly disappears into the western horizon.
You can only lie still as the last heartbeat ebbs...
You are dead! Sorry...
You receive 5 experience points.

*** PRESS RETURN ***
```

Mortal Wounds

If you are mortally wounded in combat, the skill `bandage` could save the day! However, `bandage` cannot be performed on oneself—another player has to help you. This is one of the aspects of DikuMUDs that ties players together, as different people often will help out one another when trouble strikes. If you are near death, start `shout`ing and `chat`ting that you need assistance, often others will respond. (The commands `shout` and `chat` are covered later in the section titled "Communications.")

DikuMUD Environment and Commands

As you have seen in sample DikuMUD sessions, the game is played through the use of simple one-word commands and short sentences. Navigation, object manipulation, and social interaction are all part of the gaming experience, and the Diku environment enables users to accomplish these and other tasks easily and quickly. This section outlines the most common commands and their applications encountered on "standard" DikuMUDs. Note that many DikuMUDs are heavily customized, and for this reason you should always consult a system's online help for site-specific information.

Navigating the DikuMUD World

Movement in the DikuMUD environment is accomplished simply by typing a compass direction, which may be abbreviated to a single letter. For example, walking to the room to your north may be accomplished by typing `n`. (Note that typing `north` will accomplish the same thing). Up and down can similarly be abbreviated to `u` and `d`.

Some DikuMUDs require you to move in diagonal directions, such as northwest or southeast. These may generally be abbreviated as `nw`, `se`, and so on.

Speedwalking

Some Diku systems have built-in *speedwalking* functions. This enables you to type a number of directional commands at once: `nesd` translated as `north;east;south;down`. Client programs may offer similar functionality to users of systems that do not natively support this function. Note, however, that using the speedwalking option may interfere with diagonal movement: trying to walk northeast by typing `ne` would be interpreted as `north;east` rather than the single direction northeast. Some client programs offer ways around this problem (see the sidebar "DikuMUD Client Programs" later in this chapter)—otherwise, you may need to turn off speedwalking to move in a diagonal direction. See the help for speedwalking on your individual system for directions on "toggling" this feature on and off, as it is system dependent.

help

The `help` command, issued alone, instructs the Diku system to display a comprehensive command list. `help <command or topic>` (for most commands on most systems) displays detailed help on specific commands and their functions. While this chapter can familiarize players with general aspects of DikuMUD systems, the `help` system of the player's specific MUD provides specific details on the operation, syntax, and features of that system, and should be referenced in the course of play.

look <object>

The `look` command displays the room title, description, and contents. This includes the presence of monsters, other characters, and items in the room. Note that some items or monsters may be invisible—seeing these requires the capability to detect invisibility via a potion, scroll, or spell. Also, if a room is `dark`, you will need to be holding a lit lightsource (torch, lamp, and so on) to see these descriptions.

who

The `who` command displays a list of players currently connected to the DikuMUD. Most systems display one page of names at a time, and prompt you to press the Enter key to scroll to subsequent pages. Often, `who` displays the given names, classes, and levels of the connected characters, although this is system-dependent (some systems display only the names of characters with no other information).

TIP

The `who` command is *enhanced* on some systems, enabling you to retrieve detailed information about a character. Some systems will allow you to type the following commands:

`who -n <name>`	Performs a `who` command on a specific name. This is useful when trying to see the level of a specific player.
`who <level>`	Displays a list of all players equal to or greater than the specified level.
`who -c <class>`	Enables you to list all characters of a certain class who are currently connected.

Note that these variations of the `who` command are highly system-dependent—see your system's help for detailed information.

examine <object> or *exa* <object>

Use the `exa` command to get detailed information about a specific object. `Examine` displays any set description for a particular item, and may give you an indication of its *magical* properties. *Glowing* or *humming* objects typically have some effect on the character; however, effects may be either positive or negative, and generally only apply to objects that are equipped.

You also may `exa <monster or player>` to see a list of which items a particular monster or player is using. Again, you will only see items that are equipped by the individual, and are viewable by you (non-invisible if you cannot see invisible objects, and so on). Certain character classes have the additional capability to see items in another player or monster's inventory—items that are not equipped. Typically thieves and thief-derivative classes have this capability (see the "Character Classes" section).

junk <object> or *sac* <object>

Use the `junk` or `sac` (`sacrifice`) command to destroy useless or unwanted objects. This helps keep the MUD environment clean, and also may help improve overall system performance by reducing the number of objects the MUD must keep track of. Be careful, however, as objects disposed of in this manner are *permanently* discarded. Burnt light sources, monsters, corpses, and similar items are good candidates for this type of disposal. Ridding the system of these unnecessary objects will help to free up the computer's processor, so that the game runs faster.

wear <item>

To prepare the character for combat, you must equip—or `wear`—objects in your possession. Typing `wear all` will attempt to equip all items in your inventory—you will be informed by the system what items have successfully been worn. Note that some DikuMUDs restrict certain equipment to specific character classes, or to certain experience level players. In this case, certain objects will not be `wear`able (sometimes you cannot pick up items that are too high of a level!). Worn items no longer appear in the inventory, but can be displayed by typing `eq`. You must be able to see the item to wear it (that is, if an item is invisible, you must have the ability to see invisible objects to wear it—even if you know it is in your inventory).

wield <weapon>

The `wear` command does not encompass weapons—these must be equipped separately with the `wield` command, which essentially has your character hold a particular weapon in hand, ready for combat. Like the `wear` command, you must be able to see a weapon to `wield` it.

grab <item, lightsource, or weapon> or *hold* <item, lightsource, or weapon>

The `grab` or `hold` command (depending on the system configuration) has its primary use in equipping a source of light, such as a torch. The command `grab torch` takes a torch from your inventory (assuming you have one), lights it, and moves it to your worn equipment inventory. Certain other objects may be held in hand as well, and are equipped in the same manner.

NOTE

Depending on the configuration of the specific Diku system, you may be able to `grab` a second weapon, or special items to gain certain effects or bonuses to your character attributes or statistics (*see* the "Equipment" section later in this chapter).

put <object> <container>

You use the `put` command to move objects into containers. For example, the command `put taco bag` takes a taco that is in your inventory and puts it into a bag (also in your inventory). Full, grammatically correct syntax is unnecessary—the DikuMUD assumes grammatical articles and the directional word "in"—although the sentence `put taco in bag` will achieve the same result.

remove <object>

Items are readied with the `wear` command, but may be taken off or unwielded with `remove`. This action is necessary to swap equipment (to exchange your small helm for the silvery helm, for example, you first have to remove the small helm before you can wear the silvery helm). Weapons are removed in this same fashion.

eat <food item>

You can eat edible items that are in your inventory by using the `eat` command. You will be notified by the MUD when you are fully satisfied and no longer hungry.

Food Management

It is convenient to automate the eating process by constructing an alias that both gets a piece of food from a container, then `eats` it automatically. This way, you only need to issue a single command to feed your character, and your inventory is not cluttered (and encumbered) with loose items of food. See the section on "Environment Customization Commands" for more information.

drink <drink container>

Similar to `eat`, the `drink` command tells your character to take a sip from a liquid container. Thus, to drink from your canteen, type `drink canteen` (not `drink water`!). You also may drink from stationary objects that are sources of liquid: for example, you may `drink fountain` in the Midgaard Temple Square location.

fill <container> <liquid source>

Use the `fill` *<container> <liquid source>* command to refill a liquid container such as a canteen, bottle, barrel, and so on. The most common usage is `fill canteen fountain` in the Midgaard Temple Square.

quaff <potion>

The `quaff` command is similar to `drink`, but rather instructs your character to imbibe a magical potion or flask. Whatever spells that are contained in the potion immediately will affect the character after `quaff`ing. Note that there are an abundance of poisoned potions in the DikuMUD world—`quaff` carefully!

read <readable object>

Certain objects may have writing on them, which can be viewed using the `read` command. This is most commonly used to read "MUDMail," obtained in the Post Office. Remember to `junk` or `sac` your private mail after you `read` it—do not leave it lying around for others to see!

recite <scroll> <target> or *rec* <scroll> <target>

The `recite` command is akin to `quaff`—it enables you to invoke the magic effects of a scroll. The effects, similarly, are immediate.

Scrolls of Recall

Scrolls of recall, available almost universally in the Midgaard Magic Shop, are the lifesavers of DikuMUD players. While fleeing from combat typically costs you in experience points, a scroll of recall will safely teleport you to a temple (usually the Temple of Midgaard, but this is not universal). Set the following alias for a quick escape:

```
alias rr rec recall <character name>
```

and always keep a recall in your inventory. This way, if you get into trouble, type `rr` and you will be transported to safety in a flash!

use <object> <target>

Some items can be `used`, to some effect on the character or an enemy monster, in the course of adventuring or combat. Such items typically must first be `grabbed` by the character, and then used. An example of this type of item is a wand of lightning bolts: `use wand <`*`monster`*`>` causes a lightning bolt spell to be cast on the monster. Note that many such items have a limited number of uses, after which they are rendered useless.

Social Commands

DikuMUD systems were created with the idea of player interaction in mind. To develop a sense of camaraderie and personal interaction, *social* commands are extensively supported. The "socials," as they are commonly known, enable characters to easily describe waving, shaking hands, and other personal interactions to add realism to these encounters.

<social command> <optional target>

Social commands are largely dependent on individual MUD support—there is no universal list of social or "emotion" commands (see more information on these commands and a list of available commands in Chapter 5). These commands are designed to add a bit of "personality" to MUD characters, enabling the player to yawn, tap a foot in impatience, spit, and engage in other (sometimes rude or obnoxious) behavior. Many common commands do not require a target: `yawn` causes a description of your character yawning to be displayed, while some require a target—as in `poke <target>`, which displays that you have "poked" the victim with your finger. See the online help system of your individual DikuMUD for detailed information.

Fatigue-Related Commands

To simulate the limits of characters' endurance, movement points are deducted when a character moves around in the MUD world. When these points reach zero, the character cannot move, and should `sleep` or `rest` to quickly regain movement. Additionally, characters that either are asleep or resting heal their wounds more rapidly. To resume action, a resting character must `stand`, while a sleeping character must first `wake`, and then stand up.

sleep

The regeneration of characters is much greater when `sleeping`—movement, mana, and hit points are regained at several times the "waking" rate. To put your character to sleep, simply type **`sleep`**. Sleeping characters cannot observe the events or hear the conversations around them, although many systems allow the sleeping character to use the `chat` or `gossip` commands. Additionally, many Dikus prevent sleeping players from receiving `tell` messages—the sending player is informed that the target player `can't hear you right now` when this is attempted.

wake

Issuing the `wake` command causes a *sleeping* character to become conscious, and to assume the *resting* status (equivalent to typing the `rest` command). Before engaging in physical activity, however, the character first must `stand` to get back on his or her feet.

rest

The `rest` command makes the character sit down and relax, allowing a slightly greater regeneration of movement, mana, and hitpoints than in normal `waking` mode. Although the regenerative bonus for resting is not as great as for sleeping, the resting character may observe activity in the room and take part in conversations—as well as receive incoming `tell` messages. However, the character is still restricted from physical activities, such as reciting scrolls, examining objects, casting spells, and so on.

stand

When a character is resting, issuing the command `stand` will make the character ready for physical activity by standing up. At this point, the character's rate of regeneration goes back to normal—all bonuses for resting are lost. However, the character is now able to perform all physical actions, including walking, casting spells, fighting, and so forth.

TIP

Creating aliases for resting will make the "sleeping" process easily controlled by only a few keystrokes. Some players alias the `sleep` command as the hyphen key (-), and alias the commands `wake` and `stand` as the equals key (=), shortening the process to two keystrokes.

Character Informational Commands

It is necessary to monitor the condition and capabilities of your character as you adventure in the Diku world. Informational commands will provide an overview of the physical status of your character, their skills and proficiencies, learned spells, and carried and worn equipment. All these displays contain vital data, and should be frequently referred to during game play.

score or sc

The score command roughly displays information in the following format:

```
You are a 17 year old male Human and about 1.6 m tall.
 * It's your birthday today *
You have 30(30) hit, 100(100) mana and 80(83) movement points.
You are lightly armored, and your alignment is 'Neutral.'
You have scored 2132 exp, and have 1230 gold coins.
You still have 2 practice sessions left.
You need 2999 exp to reach your next level.
You have been playing for 0 days and 3 hours.
This ranks you as Bub the Trainee (Level 2).
You are standing.
```

This display gives your age in days and hours of actual connect time, your health status, and other pertinent information. Many DikuMUDs also display a list of *statistics* on this screen, detailing physical and mental prowess of your character (see "Physical Statistics" in the "Character Attributes" section).

inventory or i

The `inventory` command displays a list of all items currently in-hand (worn items are not displayed in the inventory). Note that only objects that are visible to you are displayed—without the capability to *see invisibility*, you cannot see, manipulate, or use invisible items that are in your inventory.

equipment or eq

The companion to `inventory`, the `eq` command summons a listing of all worn equipment, including held objects and wielded weapons. Note that, like the `inventory` command, you can only see and remove items that are visible to you—if your capability to detect invisible objects "runs out" while you have an invisible object equipped, you will see the word `something` appear in your `eq` list (weapons, however, do not appear on the list).

skills or sk

The `skills` command displays the special skills and capabilities possessed by your character, including information on how accomplished you are in a particular area. You cannot use skills that are not learned—you first must *practice* with your guildmaster (see the section "Practicing and Practice Sessions"). *Excellent* rating indicates mastery of a skill, while other designations (*fair, good, poor*) denote intermediate levels of proficiency. Note that these values may be substituted for other classifications on certain Diku systems—some utilize a percentage-based system as well, where your skill level is represented by your percentage of mastery (with `100%` indicating complete knowledge and proficiency in a skill).

Skills enable characters to perform special actions that are not common to all player classes. Warriors receive enhanced attack skills that may be used in combat, thieves commonly can learn the skills `steal` and `picklock`, and spellcasters generally receive few skills (their spells offset this deficit). Unfortunately, skills are quite varied across DikuMUD platforms, so it is impossible to offer a comprehensive list of skills and their functions. However, the command `help <skill>` on most systems gives you a detailed description of the functioning of that skill and its proper usage.

spells or spe

Similar to the `skills` command, issuing the `spells` command displays a list of your known spells and corresponding proficiencies. Likewise, you must practice `spells` before you will have success in casting them.

Environment Customization Commands

The "user environment" of a DikuMUD refers to the way in which text is displayed on a user's screen, how commands are interpreted, and the appearance of text. To accommodate the varying preferences of users, many of these aspects of the game can be modified to conform to a player's individual tastes. Text often can be colored for easy interpretation, the formatting of text and status displays can be modified, and complex commands can be shortened to simple keystrokes. This section describes the commands which control this flexible environment.

toggle <setting> or *tog* <setting>

The `toggle` command (which may on some MUDs be abbreviated to simply `to`) is your "on-off" switch in the DikuMUD world. You may `toggle` communications channels (for example, `toggle chat` silences the chat channel), your capability to be `summoned` (teleported by another player with the `summon` spell) with `toggle summon`, and other settings or controls supported by your individual Diku system.

TIP

You may want to toggle a few settings right as you start playing to streamline your interface and prepare you for adventuring. The following settings are recommended:

`toggle summon`	This enables you to be summoned by other players, which may be necessary if you get in "over your head" in combat—this can be life saving!
`toggle grat`	Toggling the `grat` channel to off removes clutter from your screen, without losing valuable interactions with other players.

display <option> or *dis* <option>

The `display` command enables you to configure your prompt—the command line that the system sends you when ready for input. The prompt usually can be set to show your current hit points, mana points, and movement. While different settings are supported on various systems, the `display all` command usually sets your prompt to reflect these character statistics, giving a prompt in the form:

```
< 10 Hp 10 Mn 10 Mv >
```

Because various DikuMUDs offer other options, see your system's help on `display`.

alias <desired alias> <command to be aliased>

`alias` is one of the most useful general utility commands available to DikuMUD players, speeding player interactions with the game and drastically reducing the number of keystrokes required to accomplish a particular action. The `alias` function works in the following manner: a short "word" or set of characters are defined to represent a longer string of text. Furthermore, it is possible on most DikuMUDs to alias *multiple* commands, automating a set of related actions.

For example, you may commonly find yourself retrieving a canteen from a bag, drinking from the canteen, and then returning the canteen to the container. This process may be automated in the following manner:

```
alias h2o get canteen bag&drink canteen&put canteen bag
```

This string of commands (here separated by the `&` symbol—certain DikuMUDs may use other symbols to distinguish separate commands, so check the `help` files on your system) are performed every time the player types the much shorter command `h20`. Automation of this sort allows for easier game play and quick response—use the `alias` function to your advantage! Magic-using characters especially need the functions of `alias`, as complex combat spells can be reduced to a few keystrokes: you can substitute a simple word such as *zap* for the unwieldy command `cast 'lightning bolt'`, allowing for rapid-fire spellcasting offenses.

DikuMUD Client Programs

You may hear other players talk about using *client* programs to automate their MUDding sessions, especially the specific program Tintin++, which has been written specifically for the DikuMUD environment. Clients are terminal programs (similar to the familiar telnet), which offer enhanced features specifically geared for the MUD environment. `alias` commands are commonly supported (and enhanced) by these programs, as well as the capability to log onto several MUDs simultaneously, to completely automate actions (set certain commands to be triggered in response to text sent by the MUD), and other helpful features. For detailed information on client programs, including where to get them, how to run them, and how to make the most of their features, refer to Chapter 10, which covers "Mud Clients."

Character Attributes

Most DikuMUDs allow users to choose their race, be it human, elf, dwarf, or some other creature. Often the choice of race will impact the physical statistics of a character, which represent the physical and mental capacities of that individual. This section describes the most common races and types of physical attributes encountered on Diku systems.

Race

DikuMUDs offer players the option of choosing their character's *race* during the character creation process. Certain races (other than human, which is the standard), may impart certain bonuses—and penalties—to the physical statistics of the character. Following are the most common races:

Human: Human characters are the "standard" race against which other races are compared. Humans have a balanced set of physical statistics, without bonuses to any one stat—but also without penalties. The human race is a good choice for almost any character class.

Elf: Elf characters typically are described as being "innately magical" beings, meaning that they have a natural magical aptitude (such as high intelligence). The downside of the elf race is their frailty—typically elves have less constitution than human characters. Spell-casting characters often benefit from being elves, reaping the benefits of an increased intelligence, and, accordingly, spell power.

Dwarf: Dwarves usually are hardy folk, possessing enhanced strength and constitution. Dwarves often suffer in the area of intelligence and dexterity, however, and usually do not make effective mages. Players wishing to be fighters may consider the dwarf race a good choice for their chosen profession.

Many times, other races are made available, as well. There are no set guidelines, however, for their creation—players will have to read the MUD's specific "help" information regarding particular races. Keep in mind that races that have enhancements in certain areas often suffer in others; therefore, you must weigh the benefits of a certain race against their liabilities. The correct combination of race and class, however, can make for a very "optimized" character that is extremely effective in his or her capabilities.

Physical Statistics

The physical prowess and mental capabilities of your character are represented by *ability scores*, which typically range from 1 (abysmal) to 18 (superior). Most DikuMUDs display these "stats" in the score display, although a few use a separate listing that you can access using the `stat` command. The ability traits are as follows:

Strength: Strength represents the sheer physical brawn of your character. This stat influences the amount of damage you can do in physical combat, the amount of weight you can carry, and the size of weapon you can wield. Strength is the prime attribute for fighter-class characters.

Intelligence: Intelligence is a measure of the mental acuity of your character, which has a dramatic impact on spellcasting capabilities (the number of spell points a character has depends on the intelligence of the individual). A high score in this area results in a character learning new skills and spells more easily—the higher the score, the fewer practice sessions will be required to achieve mastery of a particular skill. Intelligence is a prime requisite for magic-user class characters.

Wisdom: Wisdom is a measure of the sagacity and intuitive powers of the character, which impacts clerics most heavily. A high wisdom is beneficial to all classes in terms of the number of training sessions gained per level—low wisdom players may receive a mere one session per level, whereas the character with an 18 wisdom might receive 5. This stat also influences the success with which cleric-class players cast spells.

Dexterity: Dexterity is the measure of nimbleness and agility possessed by a character. Highly dexterous individuals will be harder to hit in combat, and likewise, will have an easier time hitting their opponents. The capability also is of key importance in certain skilled actions, such as picking locks and stealing items—thus making dexterity the prime stat for thief-class characters.

Constitution: This stat is a representation of the fortitude of a character—the capability to resist fatigue and absorb damage inflicted in attacks. High constitution results in a much greater number of hit points (the measure of physical endurance and wellness), thus making the character better able to endure punishing attacks. Physical combat-oriented classes, such as fighters, especially prize a high constitution—although all characters benefit from increased constitution through an expanded number of movement points (the numerical representation of fatigue).

Charisma: Charisma is the measure of physical attractiveness and personality. While often berated as a "useless" statistic, it does impact the amount of money characters get for selling items in shops (for buying, as well, on some systems), and (more importantly) often impacts the percentage share of experience obtained when adventuring in groups (see "Grouping and Collaborative Combat").

Practicing and Practice Sessions

A character gains *practice sessions* when he or she advances a level. The number of sessions gained is in relation to the wisdom of the character—higher wisdom equals more sessions. The player subsequently may use these sessions to train skills and spells when in the presence of a guildmaster, by typing

```
practice <skill or spell name>
```

or

```
pra <skill or spell name>
```

While some DikuMUDs enable players to train all their skills with the same guildmaster, other Diku systems have established systems in which only certain individual guildmasters teach some skills—it then is left up to the player to seek the appropriate master from which to learn a new skill or spell.

Another consideration when training skills/spells is the intelligence of the character. The higher the intelligence stat, the fewer training sessions it will take the character to reach mastery of an individual skill. It generally is advisable for players to use all of the equipment that gives pluses to `Int` and `Wis` that they can find when leveling—this increases the benefits derived from attaining the higher level.

Hit Points

A character's *hit points* are numerical representations of the level of that individual's health. A beginning character may start with as few as about 10, whereas some DikuMUDs have level systems which allow players to eventually possess over 1,000 hit points. The greater the maximum capacity in this trait, the more damage the character can withstand (when healthy). If hit points ever reach 0, the character will fall unconscious and continue to lose points if he or she is not `bandage` by a player with that skill, or magically healed. At –10 hit points, the character dies.

The *constitution* stat determines the amount of hit points gained by a character when levels are advanced, and certain pieces of equipment (and even some weapons) may raise the maximum hit point capacity of a player when equipped, wielded, or held (this is referred to as +HIT capacity—do not confuse with HITROLL, which measures the likelihood of a character to hit with their attack).

The passage of time gradually restores hit points to an injured character, a process which is greatly enhanced when the character is `sleeping` or `resting`. Note that hit points are not regenerated continuously in this manner, but only every *clock tick*, or cycle, in the DikuMUD system. Magic spells can restore hit points instantly (this is the cleric's specialty), a great help in extended combats (spellcasters can heal themselves in the course of combat, as well as heal others). The age of the character impacts this regenerative process—the older the character, the slower hit points are regenerated.

Mana Points

While hit points represent the physical, *mana points* represent the magical powers and endurance of a character. Mana points are required to cast spells, with higher-level spells demanding greater amounts of mana to cast. Once mana is depleted, the character may not cast spells until sufficient levels of mana have regenerated. Mana points are the same as spell points on an LPMUD.

Movement Points

Movement points are a measure of your character's fatigue. They are depleted as your character moves around the MUD or takes other actions. Certain types of "terrain" may require more movement points than others—a paved road might drain one movement point per "room" moved, whereas a swamp area might cost three or more per move. Movement points are naturally replenished over time, but can be regenerated much more quickly through `sleep` or `rest`.

On certain Diku systems, movement points are utilized by the character during combat. Simply swinging a weapon (especially if it has a high weight) will "cost" movement, as will certain skills. If you play this type of DikuMUD, it is especially important to keep track of movement points before combat, as you will be unable to fight back against the monster if you run out and become *exhausted*. While your character will still recover a number of movement points at the next *clock tick*, or cycle, you regain very few movement points while you are in combat. Smart adventurers will be prepared for this circumstance with a scroll of recall—reciting the scroll when your movement points have dropped dangerously low will extricate your character from the combat (you do not require movement points to recite a scroll).

Sleeping, Healing, and Regenerating

Having your character `sleep` or `rest` will increase the number of points regenerated in all "wellness" measures—hit points, mana, and movement. Note that the *age* of a character influences the rate at which certain points regenerate. Older characters will receive more mana every *tick* (MUD clock cycle) than younger adventurers. Conversely, young characters regenerate hit points slightly faster than old. Keep in mind that certain equipment may actually raise or lower the character's age, which will affect this regenerative process.

Weapons Modifiers

Many of the weapons, pieces of armor, and items that you will come across in the course of playing a DikuMUD will be "magical," or enchanted, so that they increase your ability to hit and damage an opponent. These special enhancements are reflected in a character's hitroll and damroll, which measure the relative capability to hit and deal damage to a foe, respectively. Note that certain items actually can reduce your hitroll or damroll—these should usually be avoided.

Hitroll

The *hitroll* attribute (displayed only on some Diku systems) represents the collective bonuses of all pieces of equipment and weapons used by the character that impact the capability to hit an opponent. A high hitroll enables a character to connect with a greater percentage of blows, ultimately resulting in greater damage per round to the opponent.

Damroll

A character's *damroll* (again displayed only on some DikuMUDs) is the representation of collective bonuses to the damage done by a character's attack. A high damroll indicates that the character will do a large amount of supplemental damage in addition to the standard damage from a weapon attack—the damroll value is added to your total weapon damage for each hit. A high damroll can greatly improve the effectiveness of a character's attack—a high level character may be able to do more than 80 hit points of damage with a single stroke.

Modifying Stats

There are several ways in which stats may be modified in the course of playing a particular character on a DikuMUD. These are outlined in the following sections.

Equipment Bonuses

Certain equipment yields bonuses to certain character stats or attributes when equipped or wielded. Often, these pieces of equipment are described as *glowing* or *humming*, although certain objects which are not visibly magical in nature may confer bonuses as well. Scrolls of *Identify* will reveal the nature of equipment bonuses for a particular item—the command `rec identify <item>` uses an Identify scroll from your inventory (which then disappears) to show the special qualities of the target item.

Leveling Bonuses

A more permanent way of gaining bonuses to your physical stats is to gain levels. Levels are advanced automatically when enough experience has been accumulated, not requiring interaction or special action by the player. When a character reaches the next-highest level, there is a chance that each physical stat will increase by one. This chance (usually quite small) is enhanced by the presence of bonus-giving items being worn by the character when leveling. For example, a character who is wearing two rings that give +2 intelligence each will have a significantly greater chance to raise his or her *permanent* intelligence rating when a level is achieved, provided that the rings are worn at the time when the level is gained. Note that items in your inventory or containers *do not* count—only worn items impact the stat-raising process in this manner.

For this reason, players often trade among themselves or keep in reserve special leveling equipment, which provides poor armor value, but increases physical stats. Such items may be highly prized for this characteristic (while those that confer good armor bonuses as well are especially in demand!).

Character Classes

A character's class represents their profession, and determines what capabilities or spells are available to the player. Naturally, the types of skills and spells that are available depend on the nature of the class. Fighters receive many combat-related skills, while magic users can train in numerous spells. The following section describes the capabilities, strengths, and weaknesses of the most common character classes found on DikuMUD systems.

Fighter Classes

The prime physical statistics for fighters are strength and constitution, allowing them to overpower their enemies in pitched battle. Additionally, fighters typically have the capability to perform several attacks per round of combat, can use almost all equipment, and can take as much damage as an elephant (or dragon?).

The disadvantages of being a one-man army? No spells, few special (non-combat) capabilities, and low `int` and `wis` mean fewer training sessions for the fighter character.

Fighter-type classes are the "grunts" of the DikuMUD world—they do not (usually or to any great extent) possess spells or non-fighting special capabilities, but they compensate for this with raw muscle. Fighters generally have the most numerous and most damaging physical attacks of any character class, coupled with a large number of hit points and big movement capacities. These qualities make fighters, in effect, tanks. They deal great amounts of damage, and can withstand punishment that would make other classes run for the safety of a temple.

Low intelligence and wisdom, two common characteristics of fighters, also result in fewer training sessions per level. Thus, it may take longer to learn and master skills (although fighters do not have to worry about training spells). Offsetting this disadvantage, however, is the fact that fighters generally can wear and wield just about any item in the game (save a few), making fighters (as a class) easy characters for a player to equip. Most importantly, fighters are very well prepared to deal out, and to take, damage.

Occasionally, certain MUDs will offer alternate fighter-type classes, such as *paladins*. Paladins are a subset of fighters who follow a religious life (although they may worship good or evil gods)—evil paladins generally are referred to as "anti-paladins" and, subsequently, have access to a limited number of cleric spells. The tradeoff, however, usually is more restrictions on "wearable" equipment, and slightly diminished fighting effectiveness. Another alternate fighter-class is the *ranger*, which receives the benefit of enhanced tracking skills, some thieving capabilities, and occasionally, a limited number of spells. Like paladins, rangers typically are not as effective in combat as fighters.

If you are of the "hack-and-slash" mentality, the fighter class may be for you. With a thick metal plate, a trusty sword, and a lot of brawn, the fighter is ready for the creatures of the Diku world.

Magic-User Classes

The prime statistic of magic-users is intelligence, allowing them to cast their spells effectively. Dexterity also can be valuable, as it prevents the generally physically weak spellcaster from taking as many direct hits in hand-to-hand combat.

The greatest advantages enjoyed by magic-users are their powerful spell attacks, a rapid mastery of skills and spells with high intelligence, and their capability to use some defensive spells for physical protection.

Disadvantages of the class include physical weakness, feeble physical attacks, few hit points, and, commonly, restrictions that prevent their use of all but light armor and dagger or staff-type weapons. These liabilities offset magic-users' capabilities to inflict massive amounts of damage through spell attacks (especially at the highest levels).

The magic-user classed character is typically one of the most powerful at higher levels, with fierce magical attacks that can slay beasts that make fighters tremble. However, the difficulty comes in *getting to* high level with a magic-user—starting with few hit points (sometimes five or less!) and weak attacks, the magic-user is less effective at lower levels before offensive spell capabilities are trained.

Equipment restrictions often help physically cripple the magic-user, as certain pieces of armor usually are unwearable by this class. The most restrictive systems disallow armor of almost all types for the mage, with only robes, rings, and other miscellaneous objects allowed (although this is not the typical situation on DikuMUDs, as it is on many other systems). Weapons may be restricted as well—some DikuMUDs only allow magic-users to wield daggers and staves.

The payback for these limitations is the offensive spell power of the magic-user, as higher-level spells may outright kill many monsters. Special capabilities also are conveyed through spells, often allowing the player to fly, see invisible objects, teleport, locate objects, identify equipment, and so on. Such benefits are difficult to come by for non-casting classes.

Hybrid magic-user classes often include the likes of druids. The druid is a magic-using class that typically is identified with nature, and frequently, they receive spells that call on natural forces for their power (such as a spell that controls the weather and causes lightning to strike). Often, druids are more effective fighters than magic-users and may use more weapons and armor, in exchange for a lessening of spell power.

If you like to play it "smart," by figuring out creative combinations of spells that will aid you and weaken your enemies, and enjoy being a semi-mysterious and supernatural force in the MUD world, the magic-user is the answer to your MUD-dreams.

Thief Classes

Prime physical statistics of the thief are dexterity and strength (to a lesser degree). High dexterity is crucial in executing their special skills and capabilities.

The advantages enjoyed by the thief class include many special capabilities such as lock-picking, the capability to backstab, and other system-dependent special capabilities. Thieves generally can fight well, gain access to areas and treasure more easily and safely than other classes, and may have the capability to "peek" into the inventories of other players (and possibly even pick a few pockets!).

Disadvantages associated with the thief include a lack of spells, a tendency to be less effective at fighting than those of the fighter class, and less hit points than fighter-classed characters. While skills such as `backstab` can aid in battle, many other talents only apply outside of combat.

Thief-classed characters are not shunned by DikuMUD society as an outlaw or brigand, but rather are known for the wide variety of skills and capabilities which aid their explorations of the Diku world. While not as effective as fighters in terms of raw combat force, thieves possess capabilities that aid them in combat. `backstab`, one of the most useful, allows the thief to sneak up on enemies and stab them in the back as the first move in combat—an action which, if successful, can do a large amount of damage in that single hit. Thieves usually get extra defensive maneuvers in combat as well, with `dodge` and `tumble` being the most commonly supported skills. These techniques, which are automatically taken into consideration by the MUD in the midst of combat, may allow the thief to dodge or roll out of the way of an oncoming blow, avoiding or lessening the resulting damage.

Outside of combat, the capability to pick locks often comes in handy, granting the thief access to areas without having to first retrieve the proper key—usually from a guardian! Thieves can, as their name implies, also attempt to steal objects from monsters, or even other players (depending on the individuals).

TIP

Be sure to check DikuMUD's rules before you steal from fellow players!

A common thief sub-class is the *assassin*. Part thief, part fighter, the assassin character combines some thieving capabilities with enhanced backstabbing and combat maneuvers. This results in a character who sometimes fights as effectively as a fighter, albeit with less hit points. However, the addition of thief capabilities compensates for this shortcoming.

If you enjoy the dark alleys of your MUD, gaining access to the secrets of the realms, and the riches that are to be found, the thief class may offer you the richest set of skill options, while still giving you the capability to hold your own when it comes down to a fight.

Cleric Classes

The cleric's prime statistic is wisdom, which enables them to effectively cast clerical spells. Many clerics also possess high intelligence, rounding out their capability to learn spells quickly.

The chief advantages enjoyed by clerics include the capability to cast healing spells, knowledge of strong defensive spells, and the capability to wear more armor and wield more weapons than magic-users. This combination allows the cleric to withstand large amounts of damage, which would kill their weaker magic-user counterparts.

Disadvantages of the cleric class center around their weak offensive spells (usually much less effective than those of magic-users) and their possession of few special capabilities other than spells. Despite these drawbacks, clerics generally are sturdy adventurers well-suited to the rigors of adventuring in a Diku world.

The *cleric* is a "soldier of God," armed with healing and defensive spells and can use a wide variety of armor and weaponry. The cleric is able to both weather a fight and administer a harsh reprimand to the opponent. The capability to heal wounds stands out above all other capabilities, and can save the life of the character (as well as fellow adventurers) numerous times in the course of DikuMUD adventuring. Coupled with strong defensive spells, the cleric is an extremely formidable opponent. The disadvantages faced by clerics, however, are offensively weaker spells than those obtained by magic-users, and a lack of skills other than spells.

Cleric sub-classes include the paladin (described earlier) and similar hybrid spellcasting warriors. All these classes tend to suffer reduced combat capabilities in light of their spellcasting potential.

Players wanting to play a well-rounded character class may find their ideal in the cleric. Well defended and yet capable of making a showing in the offensive arena, capable of healing wounds in the midst of battle, the cleric is a force to be reckoned with in the Diku world.

Multiple Class Characters

Certain DikuMUDs enable players to select two or more classes for their characters, giving the resulting multi-classed character the capabilities and/or spells of *both* classes. High-level, multi-classed characters often are extraordinarily powerful—imagine the character that can deal physical damage like a fighter, cast powerful offensive magic-user spells, and heal in the middle of battle with cleric spells!

The obvious advantages enjoyed by this type of character are offset by the difficulty of achieving levels, as each separate class must be trained to the succeeding level with experience. Thus, a tri-classed character generally will require *three times* the amount of experience to advance to a numerically higher level—each class must be trained at one level, then again at the next level, and so on. For determined and committed DikuMUDders, however, there is no equivalent to the power of high-level, multi-classed characters, giving them great appeal despite the relative slowness of advancement.

Equipment

As noted earlier in the chapter, equipment must typically be *worn* or *held* to be used by a character. Certain equipment, moreover, is worn on a particular part of the body: helmets and hats on the head, gauntlets on the hands, suits of armor on the body, and so on. Only one such piece of equipment can be worn on a single location. For example, you cannot wear both a hat and a helmet simultaneously—you must choose one or the other, or swap them as your needs require (if they possess different characteristics and bonuses).

There are exceptions to the one-item rule, however. Specifically, characters can typically wear *two* of the following:

- Rings
- Amulets (neckwear)
- Bracelets (wrist armor)

In addition, a character may `grab` one object to be used as light (a torch or staff that emits light), and typically can `grab` either an additional weapon (to benefit from the hitroll and damroll or other bonuses, or also to use the second weapon in combat with a *dual weapons* skill) or a magical item of some sort. Wands, staves, and useable rods commonly are of this sort—they must be held in hand before they can be used. The types of equipment are so diverse across DikuMUD systems that individual items cannot be described here.

Another consideration regarding equipment is `saving` and `renting`. Depending on the setup of your DikuMUD, you may be able to simply type `save` and `quit` to exit the game, automatically having your items and inventory saved with your character. On this type of MUD, you will begin the next game session *exactly* as you left, holding and wearing the same equipment and possessing the same inventory.

Certain Diku systems, however, require that you `rent` your equipment before logging off. In this case, you will have to find a *receptionist* that will store your equipment before you actually leave the game—simply typing `quit` will cause you to drop everything that you are holding, saving none of your items (including worn or wielded equipment). When you find a receptionist (one is almost always present in Midgaard), you simply type `offer` to get a "quote" on how much rent will be assessed per day (real time) that you rent. Note that if you run out of money, your equipment will be sacrificed. If you are willing to pay the quoted price, the `rent` command will store your equipment and log you out of the game.

Communications

Communicating with other players is important in the DikuMUD world. Often you can obtain equipment and supplies from fellow adventurers, or enlist the aid of a group of individuals to form a "party." These activities require player communication, and the DikuMUD environment allows for many different forms of person to person and public communication.

Messages may be of multiple lines—the text is not transmitted until you press the Enter key (or carriage return on some systems).

`say <message>` sends the text message you type to every person in the room. You usually can substitute the `say` command with the `"` character.

`tell <target player> <message>` sends a message to a specific player only—it cannot be heard by other players. (The syntax for the `tell` command is `tell <player> <text...>`.)

`reply <message>` enables a player to easily send a message back to the last person who did a `tell` to them. This especially is useful in the course of a "`tell` discussion," in which players exchange multiple lines of text back and forth—the players can use the `reply` command rather than retyping `tell` and the player's name for each message.

`whisper <target player> <message>` enables you to send a private message to a specific player who is in the same room with you. However, all other players in the room will see a message to the extent that you whispered to `<target player>`. This is useful in DikuMUD political intrigue situations!

`tog <channel name>` silences the text transmitted when others send messages on the designated channel. It usually is possible to disable one or more (or all) communications channels with the `toggle` command.

`chat <message>` enables you to send messages that are seen by all other players across the MUD (except those who have toggled off the `chat` channel. The "`chat` line" is the most commonly used open discussion line on most DikuMUDs.

`gossip <message>` is another MUD-wide discussion channel, similar to `chat`. Some Diku systems offer only one or the other channel, and a few restrict their usage by making `gossip`ing and `chat`ting cost movement points, or by level-restricting the commands (brand-new players, for example, cannot use these channels on some systems before they "learn the ropes" by gaining a level or two, to keep these "newbies" from being a public annoyance).

`grat <message>` channel (the most commonly off-toggled channel on DikuMUDs) is used exclusively for congratulating fellow players on their achievements. The channel was written in to the DikuMUD environment to keep excessive numbers of this type of message (commonly found on the public channels of other MUD systems) out of the way of other discussions. It is considered to be a breach of MUD-etiquette to use the other discussion channels for this type of message.

`auction <message>` channel is just that—for use by people wanting to buy, sell, trade, or outright give away equipment. Rare and highly-valued items are commonly bid on in an auction-like manner, with the highest bidder taking home the prize. There are no official regulations for auctioning items, but common courtesy and honest bidding are the norm and expectation.

`shout <message>` enables a player to display a message on the terminal of all characters within a certain "geographic" proximity of the `shout`ing character. This especially is useful for certain collaborative efforts, in which a large group must communicate, but do not desire to use the MUD-wide communications channels. Most DikuMUDs do not allow the toggling of `shout`s—you cannot directly prevent `shout`s from being displayed.

`group say <message>` or `group tell <message>` enables a band of players to communicate amongst themselves, usually by typing `group say` or `group tell` (see "Grouping and

Collaborative Combat"). The specific syntax is dependent on the individual MUD system.

`emote <description of action>` is a special kind of communication command that you can abbreviate as a colon (`:`). It enables you to describe a character action in a third-person manner. Rather than having a message appear as `<character name>` says "...", the `emote` enables you to describe an action taken by the character in the form `<character name> <description>`. If a character named "Bob" types the command `emote looks quite bored...`, the resulting display would look like `Bob looks quite bored...` As you can imagine, this has nearly limitless possibilities if you are creative (have fun!). Note that `emotes` can be made to look like output generated by the MUD, and with a little clever emoting, you can trick people into thinking that an action or combat is taking place, when you are actually issuing an `emote`. While you should use this technique cautiously (if at all), the potential for humor (and mischief) is nearly unlimited.

Grouping and Collaborative Combat

Nearly all DikuMUDs allow characters to group with one another, each player assisting each other and sharing in the experience received for slaying creatures, and so on. This group sometimes is called a *party*, and the action of forming such a group can be called *grouping* or *partying*. This is generally accomplished in the following manner:

1. The character leading the group must `follow` him or herself. This character will do the walking for the party. Normally, the most experienced and knowledgeable player will lead (usually, but not necessarily, also the highest level player).
2. Each player to join the group must `follow <leader's name>` to start moving in conjunction with the lead character.
3. The leader must then `group <name>` for each character who wants to join the group, *starting with him or herself.* You first must be a member of a group (your own) to begin adding players to the group.

At this point, the group should be ready to adventure. Note that players may need to be re-grouped if they die, or if they `follow` themselves for some reason. Note that DikuMUDs require members of a group who are to share experience be in the *same room* when the experience is earned—in other words, a low-level character cannot join a high-level group and sit in the safety of the temple. Rather, the character must follow the others around, facing the possibility of attack by aggressive monsters and damage from "area attack" spells cast by monsters.

To communicate within the group, use the `group tell` command, which sends a message to all members in your current group (regardless of their location—you need not be in the same room to hear these communications). Issuing the `group` command by itself will display a list of all group members, their levels and classes, and their current hit points, spell points, and mana points. This can be especially useful to tell the clerical spellcasters when they should *heal* certain group members.

Individual members may leave the group by typing `follow <their name>` to follow themselves. The group leader can allow people to join or kick them out at will—`group <name>` on a player who is currently grouped will kick them out of the group. Likewise, the command `ungroup <name>` will kick out a character of the group and force him/her to stop following the leader. The `ungroup` command issued alone will disband the *entire* group.

Remember that experience is split amongst members in accordance to their level. The higher the level a character is, the closer he or she will get to receiving a "full" share (the percentage the highest-level character receives). Shares are, in addition, influenced by the charisma of the characters—if two characters were of the same level, the one with higher charisma would still receive a greater share. Some DikuMUDs also offer the `split <amount>` command, which splits the specified amount of coins amongst all members of a party—allowing the party to equitably distribute gold to the party members, as well as experience.

Finally, some DikuMUDs restrict the capability of characters to group. Some MUDs require that characters be within five levels of one another to group—other systems will impose their own rules. Consult your local help system for information.

Avoiding Grouping Restrictions

A number of Diku systems require that characters be within five levels to group. One way around this problem is for a middle-level character to be the leader, thereby allowing characters five levels higher *and* five levels lower to group. In this manner, a Level 1 character could group with a Level 10 player, provided that both the Level 1 and the Level 10 characters were grouped by a Level 5 individual.

Preparing the Character: Strategy

While most players eventually can be successful and achieve the highest levels on the majority of Diku systems, certain approaches make the process easier. The beginning player needs to play to survive and develop the necessary capabilities to meet with success later in the game, while higher level characters need to take into consideration MUD politics, in addition to seeking out the best equipment and items that can be found. This section provides some recommendations for prioritizing objectives at these different levels of play to help make your Diku experience less painful and more rewarding.

Playing for Survival: The Beginning Player

What are the most important goals for a low-level character? How do you survive in the often hard world of DikuMUD as a "newbie," or brand-new player? You must accomplish several things to succeed, and live through your first few levels.

Training

You first must train your character in a skill if you hope to be able to fight anything. While some Diku systems do not require characters to train in basic fighting skills such as `slash` and `pierce`, many of the newer systems do require these elementary skills be learned—it is not assumed that all characters know how to handle a sword, an ax, or a dagger in combat. If these skills are offered on your MUD, train a combat skill with *all* of your initial training sessions.

Do not train multiple skills initially, unless you are able to become completely proficient in a fighting skill and have extra sessions (not likely in the first level). You usually will want to train `slash`, as this skill applies to edged weapons, such as swords—the most common type. This will give you enough skill to hit the easiest monsters with some degree of accuracy (just try to hit something without training a combat skill!).

Equipment

Your second objective is to obtain equipment. Do this in any manner possible—feel absolutely free to beg higher level players for equipment and money (they expect this, and often will give you generous handouts if you appeal to their egos!). Groveling and looking pathetic may be enough to land you 50 thousand or more coins from a single player (depending on the monetary system implemented on your system), which will buy plenty of food and basic equipment. If nothing better presents itself, buy a long sword and all the basic armor in the armor shop (if your system allows you to `grab` a second weapon, buy *two* long swords).

Miscellaneous Supplies

Go to the General Store and buy a bag. Then (provided you have begged a few coins) buy about 10 pieces of bread at the bakery, and fill your bag. Buy a few torches or a lantern, too—you probably will be caught in the dark at some point. Also, do not forget to go to the water shop and buy a canteen, as you will get thirsty quite quickly. If you have managed to beg enough coins, go to the magic shop and buy a *scroll of recall* and maybe a *yellow potion of see invisible* (the yellow potion is especially important if equipment is not level-restricted on the MUD, as higher level players may give you invisible objects).

Find a Spellcaster

Once you are equipped, try to find a magic-user player and have him or her *enchant* your weapons, if the system supports this spell. Enchanted weapons will give the new player a significant advantage, and will greatly reduce the time it takes you to collect experience in the first few levels. `armor` or `bless` spells also may be helpful, and `fly` can save you from having to sleep every couple minutes to regain movement points (you won't have many). If you manage to find an especially amiable and helpful spellcaster, ask him or her to assist you for a few minutes. Spells such as `blind` and `wither strike` (these are not common to all DikuMUDs, however) can cripple monsters you normally could not kill on your own.

Find Help

Finding higher-level characters who are willing to help you will greatly ease the difficulty of your first levels. Spell assistance is the best, but simply finding a "guide" can smooth your path substantially. You occasionally will have the chance to group with higher-level adventurers, and get a share of their experience for things they kill. This tactic especially is beneficial when the player is only a few levels higher than you (you will get a larger percentage of the share than you would with high-level group members), or the players you are with are fighting *very high level* monsters and the experience value is so great that you get a meaningful share.

The Advanced Player

Do the goals and priorities of higher level characters differ from those of lower level players? The answer to this question on most Diku systems is "Yes," as the experienced and established character must confront the "politics" of the system in addition to the increased challenges of combat and adventuring. You no longer are playing to survive, but rather you compete with other players for the equipment you need to reach the highest levels and the accompanying prestige of occupying such a position on the MUD.

DikuMUD Politics

Although not universal, *clan systems* are rapidly becoming the norm on many new Diku systems. While characters inherently belong to an unchangeable class and race for the duration of their play, the clan system introduces an element of player politics into the play of the game. This addition generally adds a richness to the texture of the system, allowing higher-level characters opportunity to utilize their power in forming factions, plotting war, creating intrigue—all of which tends to enhance the multi-player aspects of the DikuMUD experience.

Clan systems typically are formed in the following manner. One (or possibly more) immortal (a wizard or "god" character, for example) sponsors the clan, providing an "advisor" who will support and council the mortal players; high-level players organize its structure, setting clan rules and hierarchy; and finally lower-level players constitute the "ranks" of the clan, bolstering strength with numbers. The purpose of these clan structures is multi-fold. First, they encourage high-level players to assist newer gamers, setting the stage for a highly interactive and supportive play environment. Second, they add a political dimension that interests and involves players on a different level—apart from the course of exploration and combat which is common to players. Political structures may become quite elaborate and involved, with treaties among clans straining under threats of "war" by other clans and their leaders. Rather than creating disunity among participants, such interclan activity generally tends to encourage player dialogue and collaboration, increasing the breadth and depth of MUD interaction.

While the highest-level players tend to dominate the political scene, you should note that this is not a firm rule. Upstart clans often begin with lower-level leaders who rapidly rise to prominence (and typically high level) due to their increased interactions with other players, and the help that they receive from clan members. If intrigue and politics are your interest, do not let the fact that you play a lower-level character interfere with your aspirations of MUD leadership.

The High-Level Player

On the most basic level, the object of the player in a DikuMUD world is the achievement of experience, and with it advanced levels. While many players enjoy world exploration and large amounts of player interaction, there is a great attraction to becoming one of the highest-level players. If you want to make this journey at a reasonably rapid pace, there are certain things you may wish to keep in mind:

Always Upgrade Your Equipment

As you advance in levels, you often will (on systems which restrict equipment by level) gain the capability to use increasingly powerful equipment. Obtaining these items should naturally be of utmost importance, but be sure to "optimize" your gear—do not "blindly" upgrade to the next level of wearable objects. Equipment often will be tailored to a specific character class, enhancing an attribute or capability important to that class. Contact other members of your class to find out what these objects are, and seek them out as you gain the ability to wear them. Spellcasters will want to increase mana, sometimes at the expense of armor class or other physical stat modifiers. Warriors, on the other hand, may want to compromise areas such as intelligence, wisdom, and mana, in return for a greatly enhanced armor class. Note that alternate sets of equipment often are carried by higher-level players, to be used in different situations (for example, one set of gear may give large bonuses to damage, where another may maximize armor class).

Be Social and Make Numerous Allies

On certain MUDs, obtaining higher-level equipment may prove difficult for the advancing character, without aid from significantly higher-level players. Additionally, increased social contact with a large number of players will greatly enhance your ability to enter into the "political scene"—especially on MUDs with clan systems or other formal political network or hierarchy. If you are viewed as a courteous and helpful player, you stand a much better chance of receiving favors when you need them. Keep in mind that characters who are a lower level than you may not always be in a position to help you now, but in the future they could be very strong. Nothing will stand out more prominently in a player's mind than an honest effort by you to help them in the past.

Form Parties

Try to group (or form parties) only with those who are close to your level when seeking your own experience points. Shares of experience will be more equitable, and you will find that larger targets may easily succumb to the efforts of a number of players. Be sure to split treasure and gold between party members, regardless of level—this will keep the group happy and encourages the idea that everyone is a significant part of the adventure. Sometimes it may not be your intention to gain experience for yourself, but, instead, for those you party with. This may be because you enjoy helping new players or you want the support of the players you are helping—for whatever reason, acts of goodwill like this are a smart idea on occasion.

Consider Using a Client Program

Client programs can automate the routine tasks commonly performed in the course of MUDding. Skilled use of such automation may make character preparation extremely time-efficient, allowing you to concentrate on adventuring and game play, rather than worrying about the basic needs of the character (food, scrolls of recall, and so on can be automatically replenished when you log on, for example). While some players consider themselves "purists" and will not use clients, the consensus seems to be that the use of such a program does not take away from game play, but rather enhances it.

Summary

The DikuMUD world offers players interactivity, collaboration, flexibility, customization, and a nearly unlimited amount of adventuring and discovery. This chapter has presented the basic techniques involved in Diku game play, but cannot adequately describe the diversity of Diku systems in all of their implementations. No two systems are alike, offering players the chance to explore many different worlds and meet dozens of people—while the fundamental concepts, command structure, and technical operation of the MUD remain similar enough to require little "relearning of the basics" for different DikuMUDs. The ease of operation and play of these systems continues to increase the user base, making DikuMUDs one of the fastest growing types of Internet games. So strap on your sword and shield, grab your armor, and plunge into the adventure of DikuMUDding.

10
CHAPTER

MUD Clients

This chapter explains the concepts and terminology associated with Internet client programs. It explores what Internet clients can do, how to set them up, and how to maximize their utility in MUD applications. Although only specific examples and commands for a small number of client programs are discussed, the general approaches and techniques introduced in this chapter are applicable to a range of client software, a number of which are briefly described at the end of the chapter.

What Can a Client Do?

A *client* program, at the most basic level, is a piece of software that handles communications with a remote system (a server). The most well-known client program is *telnet*, which enables remote access to another computer via the TCP/IP protocol. telnet establishes a connection with a remote system and allows a user to log in to that system by creating a *virtual terminal* that mimics the functions of a console actually attached to the remote computer, or *server*. MUD clients are similar programs specifically tailored to the MUD environment, offering greater capability than a simple telnet session. While this functionality varies with the individual client, the basic offerings are widely similar.

Macro Functions and Automation

A MUD-oriented client program incorporates specialized features that provide extra utility to the player. While some MUDs allow players to *alias* certain commands (also called *creating a macro*), a good client program enables the player to greatly reduce the number of keystrokes required to perform repetitive actions.

Macro Functions

An *alias* is similar to an abbreviation, allowing a user to create a substitution for a long command or series of commands, thus reducing the amount of typing required to execute the action, and allowing complex commands to be entered very rapidly. For example, if a player (here arbitrarily called Gonzo) had to type `cast 'word of healing  Gonzo` every time he needed healing in a combat, it would be very difficult to avoid errors (remember, Gonzo may be about to die, and needs that healing spell!) and would require a lot of unnecessary repetition. An alias might enable the player to set the command `heal` as an equivalent to the entire command statement, allowing much faster reaction time and less chance of error during tense moments or in the heat of an online debate.

COMMAND

Alias, or *macro*, commands allow text and commands to be "tied" to certain keystroke combinations. This typically allows for faster response, reduced typing for the user, and simplification of often-performed actions.

Navigation and Speedwalking

Movement on nearly all MUD systems operates in the same way—the player types a letter corresponding to a direction (such as `n`, `s`, `e`, `w`, `u`, `d`, sometimes `nw`, and so on) and the character moves in that direction. However, this entails a lot of typing and carriage returns (the Enter key on many systems). One feature common to many clients is the capability to *speedwalk*, or to enter multiple directional commands simultaneously, or to abbreviate commands for movement in a single direction. Using speedwalk, for example, rather than typing `w <return> w <return> w <return> w <return>` to walk four rooms to the west, you could just type **wwww** `<return>`.

Some clients enhance this capability with numerical support. You might, for example, be able to type **2wnd** to move twice west, north, and then down. This makes for much shorter movement command entries, and less confusion at the command line. The client *Tintin*, described later in this chapter, supports this function, as do many other popular clients.

Triggers and Automation

In addition to speedwalking, many clients allow certain text sent by the MUD to trigger automatic actions in response to certain input. This type of command is quite powerful—skilled use of triggers can produce a client environment that virtually runs itself, leaving the player free to worry about interacting with other players and adventuring rather than maintaining the character's constant "basic needs."

While the format of these commands will vary from client to client; their uses are equally varied—you can automate nearly any task performed in the course of interacting with a MUD. Players sometimes call extremely well-programmed and nearly "self-sufficient" client programs *robots* or "*bots*"—they literally can play the game for you. Some people argue that the degree of automation offered by client programs (robots) can take away from the gaming experience and challenge.

Programming your client program to "play the game" for you might sound like it defeats the entire purpose of MUDding. However, properly implemented automation can be quite useful even to the "purist" player who desires personal interaction over speed. There are certain circumstances when this degree of automation may be desirable. Characters often must equip themselves with basic necessities, such as buying and storing food and drink and retrieving magical items that are always in the same location. A robotic function can speed up these tasks greatly, quickly performing the actions necessary to ready the character. In this case, the client is not completely automated, but executes a series of commands that make it function nearly independently for a time. Many players set their clients to key on phrases, such as `you feel thirsty` and other informational statements from the MUD. Upon receiving this text string, the client might send the programmed response `drink canteen`, automatically quenching the character's thirst without player intervention.

Programming a bot that can actually gain experience for the player may also suit a player's purposes—many systems require a character to reach a certain level before becoming "immortal" (gaining the capability to modify and expand the MUD itself). The player who enjoys this aspect of MUD operation may want to gain experience as quickly as possible to reach this position. Others may simply appreciate the challenge of automating a character. This approach to automation attempts to control all aspects of the MUD session via actions. Movement, attacking, sleeping, and healing can all be accomplished by cleverly constructed "triggers" that continually keep the character "in action."

Text Manipulation

One useful function of nearly every MUD client is the capability to modify the text received from the remote system. Certain text may be substituted for a user-defined text string, specific phrases might be highlighted or colored (depending on the type of system the client is run on), or other filters—depending on the functionality of the particular client—may be applied to text received from the MUD. These features can help make playing MUDs both more enjoyable, and more comprehensible by suppressing superfluous or annoying text.

Gag commands typically allow text beginning with certain words or including certain phrases to be omitted from display on the user's screen. Such functionality can reduce the sometimes overwhelming amount of text input that actually is displayed on-screen—a user might set the client to gag any phrase that contains the text `leaves the room`, turning off the notification of other characters in a room. Another application in which this gag feature is useful is the "silencing" of an annoying player. If the character called `Obnoxious` keeps `shout`ing an inordinate amount of time and begins to annoy the player with a client, the player typically can gag any incoming text that starts with `Obnoxious shouts`.

Substitution commands enable the user to replace certain text strings received from the MUD with locally defined text. This may enhance the speed of the connection for users of slower systems, as well as reduce screen "clutter" by simplifying the text stream. *Highlight commands* also are available on most clients, which you can use to modify the appearance of incoming text—often bold, inverse, or colored text options are available to make key phrases stand out from the rest of the incoming text.

Custom Environment

Certain clients offer the user screen control options, allowing for color support or a *split-screen* view in which text sent from the remote system appears in one part of the screen, while text typed by the user shows up in another. While highly system- and hardware-dependent, these features enable users to customize their online environment. Split-screen features often are especially helpful to users new to MUDs, eliminating the confusion created by incoming text disrupting an unfinished outgoing command. Alternately, clients often can be configured to pause incoming text after a certain number of display lines (the *screen length*), enabling the user to read long blocks of text before it scrolls off the monitor.

Setting Up a Client

Now you move onto the specifics of installing and configuring a client for use. The following sections cover the basics of what you need to run a client and how to set it up.

System Requirements

Most MUD clients run on the UNIX platform, although a few are available for other operating systems, including MS-DOS and Microsoft Windows systems using WinSock, VMS systems, and the Macintosh. The majority of UNIX clients are written in C code, and most compile easily under BSD and System V UNIX, without extensive modification. The executable files for DOS and Mac systems are usually available precompiled, requiring no modification to the program code by the user.

Initial Setup

In order to use a client program, the source code must be acquired and uncompressed, and then finally compiled for use on a particular computer system. The easiest way to get a copy of most clients is to FTP the compressed file from archives on the Internet. On a UNIX system, this procedure is relatively simple.

1. Type **`ftp`** *`<host>`* to open an ftp session.

COMMAND

On UNIX systems, *FTP* (File Transfer Protocol) is used to transfer files from remote servers to a local computer. Note that users of different operating systems, such as MS-DOS and Macintosh, can use versions of FTP compiled for their system, or other utilities to transfer files from the Internet. See the instructions that came with your specific software for details.

NOTE

Later in this chapter (the section called "MUD Clients and Where to Find Them") you will find a list of the various MUD clients with the ftp sites and directories where they can be found.

2. Type **`cd`** *`<directory>`* to change to the directory that contains the client file. (If you do not know the exact directory, start in `/pub/` to look for appropriate subdirectories.)
3. Type **`binary`** or simply **`bin`** to tell the FTP program to prepare for a "binary" transfer.
4. Type **`get`** *`<filename>`* to download a copy of the client's compressed file.
5. When the transfer is complete, type **`bye`** to exit the FTP session.

Uncompressing the Archived File

The file retrieved via FTP is placed in a compressed format. Compression is used to reduce transmission time, disk space requirements, and to facilitate easy organization of archives. The steps you take to *uncompress* the file will vary, according to the type of computer system you are using, and the method of file compression used.

Under UNIX, if the file ends in a `Z`—meaning regular UNIX compression—you can expand the compress file by typing **`uncompress`** `<filename>`. If the filename ends in `GZ`, it has been compacted with the Gnu Utilities compression program *gzip*, and you must type **`gzip -d`** *`<filename>`* to uncompress it.

Compiling the Client Program

The first step in getting a client operational after you have the uncompressed `tar` file on your local system is to un-`tar` it. Use the command `tar xvf <filename.tar>` to restore the individual client directory and filenames in their expanded form.

`tar`, a UNIX command, enables a user to compress a number of files into a single file, to ease the transfer of the program across systems, or to a backup tape or archive. Most UNIX software (including Tintin) is distributed on the Internet in this format. The `tar xvf <filename>` command restores files from the compressed archival file, showing the user the directories and files created in the process.

The Tintin++ Client Program

Tintin++ is a client program that was originally written with a DikuMUD environment in mind (the name is said to be short for "The Kickin Tick DikuMUD Client," although how this abbreviation was arrived at is unclear). Based on variations and enhancements of an initial base of code, Bill Reiss and several other programmers put together the enhanced "plus-plus" version of Tintin, offering users flexible commands and a customizable environment. (For the sake of simplicity, hereafter, Tintin++ is referred to as Tintin.) Tintin is most popular among DikuMUD players, but may also be used in other MUD environments. Note that Tintin is a UNIX-only client—no ports have been made for other operating systems. The latest release of Tintin++, however, should compile easily under both SysV and BSD UNIX.

Obtaining the source code for Tintin should be relatively simple, as the program is archived on many different sites. You might try the archives at `ftp.princeton.edu` in the directory `/pub/tintin++/dist` as a starting point.

Refer to the previous section, "Initial Setup," for specific instructions on downloading the source file.

`source file`, which is the filename for the compressed Tintin source code, will vary with the version of Tintin that you choose to install. As of this printing, the latest version (1.5pl5) is contained in the file named `tintin++v1.5pl5.tar.Z`.

Once you have downloaded and uncompressed the `tar` file, you are ready to compile.

Compiling Tintin++

Once you have the uncompressed `tar` file on your local system, use the command `tar xvf <filename.tar>` to restore the individual Tintin directory and filenames in their expanded form.

After expanding the program files, a directory named `Tintin++` should have been created. Switch to the new directory, and read the README and INSTALL files. These files tell you of any changes to this installation procedure.

At this point you should be able to switch to the `src` directory. Type **`./configure`** to configure Tintin for your system, and then type **`make depend`**, followed by **`make`**. On most systems, this compiles the executable file, and you will be ready to play. If you have any problems with your installation or the program fails to compile, double-check your procedure. Did you get the correct file? Did you uncompress the file? Were there any error messages during decompression or file transfer? After making sure that you followed these procedures correctly, you might check the instructions provided with the source files—some slight modification to the `Makefile` may be necessary. If you are unfamiliar with UNIX and C programming, try to enlist the aid of your local "guru" in compiling the program.

Running Tintin

At this point you should have a compiled version of Tintin on your computer. The executable file `tt++` should be placed either in a user's home directory, or within the path of those users who should have access to the program (`/usr/local/bin` often is a good location).

You now should be able to start the Tintin++ program. Either change to the directory that contains the executable file named `tt++`, or place the executable within your individual path. Start the program by typing **`tt++`**. Later, you will be able to specify a file containing saved session attributes when starting Tintin by typing **`tt++`** *`<saved session filename>`*. The process of saving session defaults is discussed later in the chapter in the section "Saving Your Tintin Environment."

You now should be in the Tintin program environment. All commands you send to Tintin must be preceded with the `#` (pound) symbol. The `#help` command displays a list of all the commands supported by Tintin, and `# help` *`<command>`* displays detailed help for a particular command.

Opening a MUD Session

In order to connect to a MUD, you must use the `#session` command. The format is common to most Tintin commands—you must type **`#session`** *`{session name} {remote system and port number}`*. To connect with ELITEMUD, for example, you would type **`#session {elite} {130.237.222.237 4000}`**.

You can also type the name of a server rather than the IP number, as in **`#session {realms} {realms.dorsai.org 1501}`**. This command will connect you to RealmsMUD.

You can connect to multiple sessions (give them different session names!) and switch among them with `#ses {`*`session name`*`}`. For example, you could use `#ses {realms}` to change to your RealmsMUD session after you assigned it the `realms` session name when you originally connected in the preceding example.

Starting Points

Within the Tintin program, everything you type is sent directly to the MUD, with the exception of lines that begin with a `#` symbol. The pound sign (`#`) is called the *command* character, and tells Tintin to expect a command, rather than text, to be sent to the remote system. You can change the default command character from within Tintin with the `#char` command, with the following syntax:

`#char` *`<character to become command character>`*.

The default command character is `#`, but you can change it using the `#char` command. Alternately, you can set the default command character for a certain saved session, as the first character read by Tintin of the `save` file will be set as the default command `char`. The following section discusses saving and saved files.

Saving Your Tintin Environment

After you configure the Tintin session, you can save your aliases, highlights, and so on, to a configuration file that you can load for future use. The following two commands make managing the saved environment easy:

`#write <`*`filename`*`>`	The `#write` command saves your current environment to a configuration file that you can open in a subsequent session to access all your aliases, actions, highlights, and other session parameters. Note that if the specified file already exists, it will be overwritten.
`#writesession <`*`filename`*`>`	Using `#writesession` copies all your current modifications to the active session to a previously written configuration file. If the specified file does not exist, it is created.

MUD Navigation with Tintin

The Tintin client offers users functionality that allows for easy navigation of a MUD environment. The speedwalking and path functions both reduce the number of keystrokes required for such navigation tasks, and make the process quite rapid.

Speedwalking

The *speedwalking* function provided by Tintin can greatly speed MUD navigation. Rather than typing a single direction each time the MUD presents the user with a prompt, the speedwalk function allows the entry of numerous directions, which will be sent to the MUD in rapid succession. When speedwalk is active, you may type a string of commands that specify the number of times to travel in a certain direction. For example, the command `4n2euw` is identical to sending `n, n, n, n, e, e, u, w` to the remote system in rapid succession.

By issuing the Tintin `#speedwalk` command, the user can turn speedwalking mode on or off. (The speedwalk feature is either on or off—the command simply toggles between the two and requires no argument.) With speedwalk off, the client will not interpret movement strings, as described previously. This can come in handy if you need to send a set of characters to the MUD that normally might be interpreted as movement, but instead serve some other function.

The *path* Function

Along with the speedwalk option, *path settings* are designed to facilitate the rapid movement of the character through a MUD environment. The `path` function, in conjunction with speedwalk, keeps track of your movements through a MUD, and allows for saving of these paths and automatic return to a starting point. The following commands control the functioning of this feature:

`#mark`	This command clears any previous path data, and marks your current location as the start of a new path.
`#path`	The `#path` command displays your current path data.
`#unpath`	This function deletes from the path the last move you made.
`#map <direction>`	This command enables the user to arbitrarily add a direction to the end of the path.
`#save (#savepath)`	These commands (`save` is an abbreviation for `savepath`) enable you to save the current path as an `#alias`. The character then could execute the same movements in a later session; you simply type the name of the `#save` alias, and the same string of directional commands will be executed. Provided that the starting point is the same, you will be able to return to a specific location very rapidly via the saved path.
`#return`	The `#return` function "walks" a character back to the starting point of a path, by removing the last command from the top of the path and reversing the directions. This is extremely useful for exploring—you will not get lost!

The *#alias* Command

One of the most widely used functions of the Tintin program is the `#alias` command. With it, a short word or abbreviation can be defined to represent a longer string of text, or a number of separate commands. This can be very useful for often-repeated commands, reducing lengthy phrases to a small number of keystrokes.

The `#alias` command, given without other arguments, displays a list of all aliases currently defined. Alternately, you can obtain a limited listing of aliases using the wildcard character `*`, which denotes any string of text. The command `#alias a*`, for example, displays all defined aliases that begin with `a`.

Variables within Aliases

It also is possible to include variables within aliases, using the format `%<0-9>`. If a variable is included in an alias, additional text will be added to the aliased statement when the alias is executed and given an argument—`%0` is set to all the text after the aliased word, `%1` is set to the first word after the alias, `%2` the second word after the alias, and so on. To give an example, the alias `#alias stare emote stares at %1 in disbelief.` could be executed with the following string of text:

```
stare George
```

This, in turn, would (on most MUDs), display the following:

```
<your character name> stares at George in disbelief.
```

The Greatest Life-Saving Alias

When playing combat-oriented MUDs, always set an `alias` to recite a scroll of recall on yourself. The alias `#alias rr rec recall <character name>` has saved more than one adventurer's life!

Alias Examples

Certain commands and actions are commonly aliased by MUD players. The following examples represent just a few of the most popular, and most convenient, aliases.

```
#alias cure cast 'cure' %1
```

Typing `cure Bob` would send `cast 'cure' Bob` to the MUD.

```
#alias dc drink canteen
```

Typing `dc` sends `drink canteen` to the remote system.

```
#alias {eat} {get taco bag;eat taco}
```

In this example, typing `eat` would send both `get taco bag` and `eat taco`.

You also can imbed other Tintin commands within an alias, as in the following:

```
#alias {goelite} {#ses {elite} {130.237.222.237 4000}}
```

Typing `elite` causes Tintin to attempt to connect to ELITEMUD. Note that you must match your braces—each pair of braces (which must be used to separate arguments of multipart commands) must be completed, or the statement will result in an error.

Aliases and Spells

When playing combat-oriented spells, always `alias` your spells (or your special abilities if you play other classes) if you are a spellcaster. Short words such as `fry` are easy to remember, and are much quicker to type than `cast 'lightning bolt'` while in the middle of combat. In addition, you may want to `alias` an entire series of spells that commonly are applied to a single player (or yourself) during play. This is especially useful for protection spells, as you can easily design an alias `protect` that will cast a series of defensive spells on you.

Verbatim Mode

By default, Tintin interprets, or *parses*, all text that is typed by the user. That is, every time text is entered and a return character is detected, Tintin checks to see if the typed text contains something the user has set as an alias. Should text need to be sent to the remote system, unchanged or without translation by Tintin, a single `#` followed by a carriage return will tell Tintin to send text "as is," or *verbatim*. Another single `#` will end this verbatim text mode (the `#verbatim` command achieves the same effect).

`#verbatim`, which you can abbreviate to a single `#` symbol, toggles verbatim mode on and off. In verbatim mode, all text is sent directly to the remote system, without parsing.

Removing Aliases

The `#unalias <alias>` command removes an alias from the active Tintin session. Note, however, that using a wildcard in the `#unalias` command results in the removal of only the *first* match, not all matching aliases.

Automation: The Tintin *#action*

In the course of playing combat MUDs, frequent users will soon notice that many actions are repetitive and time-consuming. Actions for eating, drinking, and responding to certain situations encountered during game play require a great deal of typing, and may detract from the playing experience itself. By using the `#action` command, the Tintin client enables users to circumvent certain repetitive actions that are triggered by set strings of text. A user may easily set the client to search for specific strings of text from the MUD, and if that string is received, a certain command (or commands) is executed without user prompting—without so much as a keystroke.

`#action {string} {command(s)} {priority}`: The `#action` function sets Tintin to scan for a text `{string}`, and when that string is received, the `{command(s)}` are executed. Priority may be from 0-9 (5 is the default if not specified), which establishes the `#action` to which Tintin responds in the event of multiple triggered actions. Actions set as priority `0` are executed first, while priority `9` go last. You can substitute variables (`%0-9`) from the input string to be used in the command side of the `#action`, as in the following example:

```
#action {%1 pokes you.} {poke %%1}
```

If the text `Bob pokes you.` is received, Tintin automatically issues the `poke Bob` command.

Auto-Tracking

On MUDs that support a "track" feature, which indicates the direction to a certain target name, Tintin actions can be easily designed to do the "walking" for you. First, determine the format of the track messages for a particular MUD—many use the form `You sense a trail <direction> from here!`. Then set `#action` commands for each direction. Use the following format:

```
#action {You sense a trail north from here!} {n}
```

Repeat this command for each direction (remember up and down!), and you should be all set for hands-free tracking. Issuing the `track <name>` command should then send you speeding on your way to the target—but watch out! Because you do not have direct control over your movement, the auto-track may take you into dangerous parts of the MUD. (Do not auto-track near deathtraps!)

Customizing the Tintin Environment

Another advantage of running a MUD client program is the capability to customize the appearance of your screen. Tintin supports a number of functions that enable users to add color and highlighting to incoming text, to split the screen into two viewing areas, and even supports features that can radically alter or suppress certain portions of the text stream. This section demonstrates how these functions can be used to enhance the playability and readability of a MUD session.

Highlighting Incoming Text

Users are given several options for `{type}`, which specifies the manner in which the text should be displayed. Valid options include `bold`, `red`, `blue`, `reverse`, and many others. See `#help highlight` in Tintin for a complete list.

`#highlight` enables you to request Tintin to alter the appearance of incoming text (text sent by the MUD). The following format is utilized:

```
#highlight {type} {string}
```

Pruning the Incoming Text Stream

Often, incoming text from a MUD can be extremely verbose, quickly filling the screen with extraneous messages. The `#substitute` (`#sub`) command can easily remedy this situation by eliminating specified incoming text strings, and replacing them with different, user-defined messages. This is accomplished in the following manner:

```
#sub {text} {new text}
```

Note that if a . (period) is the only character specified in the `new text` parameter, the line will simply be deleted. This is commonly referred to as *gagging* text.

The following are examples of substitution:

`#sub {Bob says} {.}`	This command causes Tintin to suppress (`gag`) the display of any lines starting with `Bob says`.
`#sub {^Bob%0} {BOB%0}`	This line tells Tintin to replace all lines that start with `Bob` (due to the presence of the ^ character) with a capitalized `BOB`, appending any text that follows.

To remove a substitution, use the command `#unsub {text}`. The specified text no longer will be substituted.

Splitting the Screen

Provided that you are running Tintin on a vt100 or ANSI-compatible terminal (or emulator), Tintin gives you the option to split the screen into two areas—one which displays text typed by the user, and the other for text received by the MUD. This can be extremely useful on busy systems, where typed text often is interrupted by rapidly incoming text.

`#split {line #}` divides the screen into two sections, with the top displaying incoming text and the bottom echoing a user's input. The `line #` specifies at which point on the screen you want Tintin to make the split.

This flexibility enables users with various-sized screens to split at a line they find convenient. The `#unsplit` command returns to regular display, without the split.

Advanced Functions

While many users use Tintin only for its aliasing and text-handling capabilities, the program contains several additional features that give it considerable "programming" power. Conditional operators can specify conditions for automated actions, mathematical calculations may be performed, and variables can store the results of computation. Properly utilized, such functions give users the capability to create elaborate automated systems.

Conditional Operations with *#if*

The `#if` command is one of the more powerful commands offered by Tintin, allowing the configuration of elaborate `if-then` type operations embedded within other commands and macros. Essentially, the `#if` statement evaluates a set of statements, and if the result is "true," then the following command is executed. For example, consider the following statement:

```
#action {^< %0hp} {#if {%%0<=100} {flee}}
```

This `action` command is first triggered by a text string indicating the player's hit points, at which time the `#if` statement is evaluated to see if an action—in this case the MUD command `flee`—is executed. The client will check the value retrieved by the `%0` variable, and if the value is less than or equal to 100, execute the `flee` command. Expressions may use the same operators as the `#math` command, enabling you to evaluate numerical values in the conditional statement.

Mathematical Operations and Variables

Tintin allows users to define variables to be used in output or evaluation of conditional statements, giving the user flexibility in designing a customized interface with context-sensitive responses and actions.

`#math` stores the result of an expression in a variable, which then is usable by other Tintin commands. The command takes the form: `#math` *`<variable>`* *`<expression>`*—the expression is evaluated in the same way that the conditional portion of an `#if` command is resolved, but rather than returning a "true" or "false" result, the actual calculated value of the expression is stored in the specified variable.

Consider the following example:

```
#math {missile} {$mana/5}
```

Assuming that you already have a variable named `$mana`, this expression will take the value of the `mana` variable, divide it by five, and store the result in `$missile`. A player could then set up a command such as:

```
#alias {zap} {#$missile cast 'magic missile'}
```

This `alias`, when activated by the `zap` command, causes the spell `magic missile` to be cast as many times as possible with a player's available `mana`.

NOTE

Tintin evaluates conditional or mathematical expressions in a manner similar to that of UNIX, and likewise can accept a range of logical operators, including the following:

Operator	*Function*
!	Logical not
*	Multiply integer
/	Divide integer
+	Add integer
-	Subtract integer
>	Greater than (result is non-zero or zero)
>=	Greater than or equal (result is non-zero or zero)
<	Less than (result is non-zero or zero)
<=	Less than or equal to (result is non-zero or zero)
= or ==	Equals (result is non-zero or zero)
!=	Not equal to (result is non-zero or zero)
& or &&	Logical and (result is non-zero or zero)
| or ||	Logical or (result is non-zero or zero)

An expression is considered `True` if it is any non-zero number, and `False` if it is zero. Expressions within parentheses have the highest priority, and are evaluated first.

The *#loop* Function

The `loop` command gives the flexibility of performing a single operation on numerous objects successively, incrementing a numerical value by one for each "pass" of the operation. Similar in function to the programmer's `for-next` loop, you also can use `#loop` to successively decrement a value for each iteration of the statement. You might create a command that uses the `#loop` function to pick up all the objects from three different containers, each specified by a number. In the proper syntax, this looks like the following:

```
#loop {1,3} {get all %0.bag}
```

This is exactly the same as issuing the commands `get all bag; get all 2.bag; get all 3.bag`, but significantly more streamlined and compact.

If the first value specified in the numerical arguments has a greater value than the second, the loop will decrement, as in the following:

```
#loop {4,2} {drop %0.ring}
```

This command produces the result of dropping `4.ring`, then `3.ring`, and finally `2.ring`.

Note that you can embed the `#loop` function within other Tintin commands. You could, for example, create the following alias:

```
#alias {3keys} {#loop {1,3} {get key %0.corpse}}
```

By issuing the `3keys` command, you would retrieve keys from the first three corpses in the room.

Executing System Commands within Tintin

The `#sys <command>` command enables you to execute a `shell` command from within Tintin itself. This enables you to back up configuration and similar files without leaving the client environment.

You also can embed a `#sys` command within another command, such as `alias`, giving you the option to set up automated routines that back up your configuration files or perform other tasks in the course of a MUD session. The following `#alias` will make a backup of your save file before actually performing a `#write`:

```
#alias {backup} {#sys cp newfile oldfile:#write newfile}
```

This routine copies the contents of the file named `newfile` to a file called `oldfile`. After this operation, the current session settings overwrite `newfile`.

A Macintosh Client: MUDDweller

While the majority of MUD clients run under the UNIX operating system, client programmers are beginning to address the needs of users who run from other platforms and lack access to a workstation. *MUDDweller*, a program written by Oliver Maquelin, provides basic client functionality for the Macintosh operating system. Allowing connections through either the Mac communication toolbox or MacTCP, the program supports multiple sessions, a command history, and an integrated file transfer system among other common client features, such as aliasing.

MUDDweller was developed specifically for use with LPMUD-type systems, offering LP Wizards the capability to up- and download files from their system via the client program and the ed program on the remote MUD system. However, the program is easily adapted to other MUDs as well. MUDDweller provides a line-oriented terminal emulator interface for connections to the remote MUD, with additional functionality to enhance MUD interaction.

Configuring MUDDweller

The initial step in setting up a MUDDweller system is to obtain a copy of the compressed source for the program—many MUD-related Internet sites maintain archival copies of the program (often in `misc.` sections under client programs, as the majority of MUD clients are UNIX-based). Depending on your configuration, you might use either a modem terminal program to download a copy of the file, or use an application such as *Fetch* to retrieve a copy over a TCP/IP network if MacTCP is installed. See the documentation of your transfer program for specific instructions.

The downloaded file will be in a self-extracting format; double-clicking on the program icon should start the decompression process. You should eventually end up with a MUDDweller folder containing the client program and documentation files.

The first time MUDDweller is run, you will need to configure the program for your local setup. Several aspects of your session must be defined, and you have the option of modifying a number of default settings.

Connection Type

MUDDweller gives you the option of making a connection to a remote system using either the Apple communication toolbox or the MacTCP (TCP/IP) driver. You may select one of these settings under the Configure menu in the Communication section when the MUDDweller program is started.

NOTE

If you have an autodialing SLIP or PPP protocol driver installed on your Mac, starting the client program will initiate a dialing sequence. If this is the case for your system, be sure that your computer is properly configured and connected to a phone line before starting the MUDDweller client.

The connection subsequently may be configured with either the Connection or the TCP/IP Address menu items, depending on the connection method selected. The Connection option enables you to select the desired serial connection tool, and appropriate settings for your network configuration—this varies by system. Alternately, the TCP/IP setup screen will prompt for the IP address (or machine name) and port number of the desired remote system.

Preferences

Several program environment settings are user-customizable under the Preferences menu.

Screen font	This setting enables you to specify the screen font displayed by the client program.
Tab width	This sets the number of "spaces" a tab represents—pressing the Tab key will advance the cursor this many spaces.
File type	Changing file type alters the default format for text captured in a session log.
Log size	This option enables you to specify the maximum size for a session log, expressed in kilobytes.
History lines	The History lines setting determines the amount of text that is retained in memory, and will be available for the user to scroll back and view.

Communication

In the Communication dialog box, the user is offered the option of setting communications defaults, including the interpretation of end-of-line characters, whether to use a standard telnet protocol, vt100 terminal emulation option, local echoing status, and whether to ignore carriage returns received from the remote system. The program's default settings are ideal for TCP/IP setups (such as direct Internet connections), but may need modification if you are using the Communications toolbox to connect to a remote system. This type of setup often requires only carriage returns (CR) to be sent at the end of a line, and local echo may be necessary as well so that text typed by the user is displayed on the "local" screen.

File Transfers

The File Transfers configuration screen presents the configuration options for MUDDweller's file transferring function. You can select either MTP or ed-based transfers, as well as default options, such as the upload directory and the procedure for tab conversion. See the section "Sending and Receiving Files" for additional information about these settings and their functions.

Saving Your Configuration

You can save your newly configured MUDDweller session with the `save` or `save as` command, which is found under the File menu. You can load these settings under subsequent sessions of MUDDweller, so that reconfiguration is not necessary each time you execute the program.

Opening a Session

After initially configuring MUDDweller, you should be ready to connect to a remote MUD system. Remember that if you are using MacTCP, the session is specified under TCP/IP Address in the Configure menu item. To open this session, choose Open Connection from the Configure menu. This causes MUDDweller to open a session at this IP location. If you previously have saved your configuration, pressing and holding down the Option key during MUDDweller's startup will instruct the program to prompt for the name of a configuration file, rather than starting up a new session with no previously saved defaults.

The User Environment in MUDDweller

Once a connection is established, the MUDDweller environment will split the text window—the top will display text received from the remote system, and the bottom will display locally typed text. Note that MUDDweller is *line oriented*; that is, text you type is not sent to the MUD until you press the Return key. You can resize the input and output windows simply by clicking and dragging the separator bar between the top and bottom screen segments. Additionally, you can resize the entire text window to fit your display—click and drag a lower corner of the program window to perform this function.

Logging a Session

MUDDweller offers the option to capture the entire text of your session, including user input and text sent from the MUD, to a file. Choose Log to File... from the Configure menu, and then select to specify the filename and location of this captured data. Subsequently choosing the Close Log option from the Configure menu will end the logging process and close the file.

NOTE

When connecting through the Communications toolbox, a `send break` command is available under the Send pull-down menu. This command sends a "break" signal to the remote system, where the function of this signal will depend on the nature of the communications tool being used. This function is not available and is dimmed when using MacTCP. This command can be used by certain programs to exit remote processes that have ceased to respond, and other similar "abort" functions. See the documentation that accompanies your specific connection tool for details on this type of function.

Sending and Receiving Files

Another useful feature of MUDDweller is the capability to transfer files to and from a remote system via the client program. This function was tailored to suit the needs of LPMUD "wizards"—the players responsible for the maintenance and development of an

LPMUD, who have access to the files that comprise the MUD "world." MUDDweller offers the option of using either MTP protocol transfers, or using the ed program to import and export files.

The MTP (MUD Transfer Protocol) protocol was written by the player Mentar of TUBMUD. It requires that an MTP server is running on the remote system, and provides a direct protocol for the exchange of files between your system and the MUD. Further, it is only usable under the MacTCP protocol—you cannot use MTP through a Communications toolbox connection. If this system is installed on your remote MUD, however, you can directly upload and download files through MUDDweller's `upload file` and `download file` commands offered under the Send menu (assuming that you have appropriate access on the remote system, of course). LPMUD wizards will find this function useful for creating areas and objects offline, and having an easy way of installing them on the actual MUD system at a later time.

If your system does not support the MTP protocol, you still may have the option of transferring your files easily through MUDDweller—provided that the MUD system runs a standard version of the ed editor. By specifying the ed transfer option in the File Transfers section of the Configure menu, you can instruct MUDDweller to transport files by sending the appropriate commands to ed and printing lines of text to the remote file. You also can download files using this technique. While not as fast as the MTP option, this transfer mechanism offers compatibility with the majority of LP-based MUDs. You also can instruct the client in the file transfers setup to send an update command to the remote system, and interpret tabs as a certain number of spaces for conversion purposes.

The Update and Full Update commands under the Send menu instruct MUDDweller to transfer entire directories of files to the remote system. Update sends just those files that are new or modified since your last Update—a list of the files updated and the corresponding dates of the updating are stored in your session file. Full Update transfers all files in the default transfer directory, regardless of the modification status recorded by MUDDweller. Note that you may halt the update procedure using *`command`* . (period), or by clicking Stop in the send update dialog box.

Be careful when using the Update command—if you have accidentally set the update folder to your "desktop " because this will cause MUDDweller to attempt uploading the *entire contents* of your hard drive when you execute the Update command! Be sure that you specify a folder that contains *only* MUD-related files that you want to automatically update with the client program.

Macros

Macro commands give MUDDweller users the option of defining short keystrokes to represent long, complex or often-performed actions. The `login` and `logout` macros are invoked automatically when opening and closing a remote connection, and additional macros can be assigned to any combination of the Shift, Control, or Option keys, along with an alphanumeric key.

To create a new macro, choose Macros from the Configure menu. You can edit previously defined macros by selecting them from the Macro: name window and then editing the macro commands in the text box. You also can remove the current macro by clicking Remove, and save changes by clicking Done. Clicking New enables you to create a new macro—you first will be prompted for the key combination you want to define. After selecting New a blank macro is added (described in the Macro window as your chosen key combination)—you now can edit the commands of this newly created macro in the text window, and finally save by choosing Done.

Macros consist of one or more lines of text, each line containing only one command. Blank lines are ignored by MUDDweller, as are any lines beginning with the `#` character. Commands execute sequentially until the end of the macro text, or a user interrupt (`<command> .`) is received. The command line must begin with one of MUDDweller's defined macro commands, which, in turn, usually is followed by certain parameters or text. The following commands are supported:

`echo <text>`	This command displays the specified text to the user's screen—nothing is sent to the remote MUD system. Note that a carriage return is *not* automatically sent at the end of a line—you must specify this action by ending the text string with a `\n` character combination.
`match <text>`	The `match` command instructs MUDDweller to wait for the specified text to be received from the MUD—operation of the macro halts until this time. Note that the text must match exactly, as there currently is no wildcard support.
`passwd`	This command prompts the user to enter a password, which, in turn, is sent to the MUD. A command such as this is useful when designing automated login sequences that do not store the actual password of the user. Note MUDDweller does not display the typed password.
`quiet`	The `quiet` command toggles "quiet" mode, where no output is displayed in the main window except the output of `echo` macro commands (which always is displayed). You may find this feature useful in automated login routines to hide user names and passwords. A second command, `quiet off`, returns the display to normal, and MUDDweller automatically reverts to normal display mode at the end of the macro.

`send <text>`	This command sends the specified text to the MUD. As with `echo`, a carriage return must be specified with the `\n` notation.
`wait <number of seconds>`	The `wait` command halts execution of a macro for a specified number of seconds, at which point the macro will continue with the next command line.

Special Characters

In addition to the carriage return (`\n`), you can include other characters in macros as well. The `\t` sequence sends a <tab> character, `\f` sends a form feed, and `\b` generates a backspace. Additionally, you can produce special character codes by typing `\` (backslash) followed by an octal digit. The most important of these is `<control> c`, which is generated by the `\3` notation.

Another Client Program

Dozens of client programs are available via the Internet, each providing numerous options for MUD players. Some, like Tintin, were designed with particular MUD systems in mind, taking into account the special needs of their users. The most outstanding features of one of the more enduring and popular programs, TinyFugue, are discussed in the following sections. While TinyFugue is an older program, the robust list of features and the flexibility of the client maintain its popularity with the online community. TinyFugue was developed with users of MUCK and MOO systems in mind, and is quite adaptable to the demands of a social-MUD environment.

TinyFugue

The *TinyFugue* system has long been a popular client interface for players of MOO, MUCK, and many TinyMUD-derivative systems. With a robust feature list supporting multiple sessions, macros, triggers and automation, command history and other functions, TinyFugue offers users maximum control over their environment. Although more recent programs such an Tintin++ have gained large followings, many MUD players continue to use TinyFugue because of its power and flexibility in the hands of an experienced client programmer.

Loading "Worlds"—Session Management

TinyFugue provides commands that allow users to manipulate several remote MUD sessions simultaneously. Following are some basic commands you should know:

/addworld:	allows adding a system to your list of known MUDs, using the format **/addworld <session name> <host> <IP> <associated macro file>**
/unworld:	remove a world with this command
/world	**/world <name>** will attempt to make the specified world active
/loadworld:	this command will load the defaults for a specific world
/saveworld:	you may save a particular world's defaults with /saveworld <name>
/listworlds:	this command lists all currently defined worlds

Macro Functions and Automation

TinyFugue allows users to define complex macro functions, complete with optional associated triggers and execution probabilities. Macro functions can be defined in the following basic ways:

`/def <name> = <body>`

or

`/def [-1[-]] [-p<priority>] [-c<chance>] [-t<"pattern">] -f<function>] [<name>] [= <body>]`

The `def` command provides a powerful interactive tool to control and manipulate the MUD environment. At the most basic level, you can use this function to define an alias—the command `/def ga = get all` would alias `get all` as `ga`. However, as the second format shows, there is a whole range of user definable options, allowing a priority and percentage chance to be included in the automatic functioning of an automated response to incoming text (whew!). This obviously is more complex, but offers significant flexibility and high degrees of customization to the environment.

`/trig <"pattern"> = <body>`

The `/trig` command provides for session automation—the client executes specified commands upon receiving certain text from the MUD. A derivative of the `/def` command, you can customize triggers to include information about percentage chance of execution, priority, and so on. The TinyFugue online help displays a number of useful examples for constructing automated actions of this complexity.

Additional Information—TinyFugue

TinyFugue is a powerful client program, offering many additional functions and options that enable you to create a truly custom environment, tailored to the needs of a particular system or group of systems. More detailed information can be found in the distribution package which is maintained at the following sites:

`beta.xerox.com` in the directory `/pub/MOO/contrib/clients`

`ftp.tcp.com` in the directory `/pub/MUD/Clients`

Glossary of Client Terms

What follows is a quick summary of terms you might encounter while discussing clients with fellow players or while reading the documentation on whatever client you choose to use. Thanks again to Jennifer Smith for permission to use these from the MUD FAQ.

Auto-login—Automatically logs into the game for you.

Highlighting—Allows boldface or other emphasis to be applied to text. This often is allowed on particular types of output (such as `whispers`), or particular players. `Regexp` means that UNIX-style regular expressions can be used to select text to highlight.

Gag—Allows selected text to be suppressed. The choice of what to suppress often is similar to highlighting (players or regular expressions).

Macros—Allow new commands to be defined. The complexity of a macro varies greatly among clients; check the documentation for details.

Logging—Allows output from the MUD to be recorded in a file.

Cyberportals—Supports special MUD features that can automatically reconnect you to another MUD server.

Screen Mode—Supports some type of a screen mode (beyond just scrolling your output off the top of the screen) on some terminals. The exact support varies.

Triggers—Supports events that happen when certain actions on the MUD occur (such as waving when a player enters the room). (This can nearly always be trivially done on programmable clients, even if it isn't built in.)

Some of these clients are more featured than others, and some require a fair degree of computer literacy. TinyTalk and TinyFugue are among the easiest to learn; Tcltt and VT are more professional. Because many MUDders write their own clients, this list is constantly changing, so be sure to ask around.

Summary

Clients can be incredibly useful and offer a diverse range of functionality to MUD players. Because anyone with a knowledge of programming can write a new MUD client, the number of clients changes often. Also, because many of the existing clients include their source code on FTP sites, there are many modified versions of the popular clients. Tintin, for example, also is available in a version called *Tintin++hacked*, which is slightly different from the original version of Tintin and contains some improvements to the code.

A client can be very useful tool for making MUDding easier and more efficient. This chapter has only presented the rudiments of client use and programming—the limits of their usefulness are dictated solely by the limits of peoples' creative application. Simplifying a user's interface, organizing the display of information, and reducing keystrokes of the client is an incredible boon for avid MUDders. In the hands of a clever programmer, a client virtually can become robotic, performing with little or no user input, and possibly even fooling other players into thinking that it is a person and not a programmed machine. In either extreme, the client is a powerful tool that MUD users should not be without.

II

CHAPTER

Being a Wizard (MUDding at the Next Level)

The concept of a *wizard* means many different things on different types of MUDs. But as a generalization, it usually is used to designate a player who has been granted certain privileges beyond the normal player. These privileges often include the capability to build or program areas (if that power is not given to everyone), the capability to take action against abusive players, the capability to alter or approve regions of the MUD, or even the capability to delete, modify, or snoop on players. Some of these powers are discussed in detail in this chapter, but the next section of this book is devoted exclusively to programming and wizard functions with chapters on the specific types of MUDs. This chapter is more for the user who wants to be familiar with the powers wizards have, what being a wizard is about, and how to become a wizard.

What is a Wizard and Why Would You Want to Be One?

In this chapter, wizard is used as a generic term. A wizard could be called a god, elder, administrator, janitor, builder, staff, or any of a number of other possible titles. On most MUD systems, however, there is a least one class of player that carries the title "wizard."

Wizards are super-players or super-users. They have powers beyond those of a normal or *mortal* player. In fact, on combat MUDs one big advantage of being a wizard is that you are *immortal*—nothing can kill you. Being a wizard certainly has its privileges—being immortal certainly is a nice perk.

Some general wizard powers include the following:

- **Unrestricted teleportation**—The capability to teleport is available to players on some MUDs (especially MOOs) as a general power and sometimes as a spell within certain guilds. This type of teleportation usually works in only one way. For example, I could open a gate to someone else and they could step through it to me. Sometimes MUDs even allow teleportation directly to a player (this tends to be disastrous on MUDs that allow player killing). Wizards usually can teleport a player directly to the wizard's location or teleport directly to any player or room. Wizards don't need a special spell for this.
- **Building or Programming**—The capability to build new rooms and program new objects often is restricted. On LPMUDs and DikuMUDs, only wizards have this power. On MOOs, MUCKs, and MUSHes, everyone is allowed this power, but to add new rooms they typically need approval—usually by a wizard. Wizards are responsible for the creation of new areas on MUDs and for the organization and geography of the MUD world.
- **Snooping**—Snooping enables wizards to, well, *snoop* on other players. They see everything the snooped player sees and does. This is very useful for debugging and checking on players who have been accused of harassment or cheating. It is obvious, however, that this power is one that can be easily abused. It often is restricted to only the highest level wizards. Abuse of this power will likely cost you your wizard status. This power primarily is found on the LPMUDs and DikuMUDs. However, on MOOs, MUSHes, and MUCKs—as well as on LPMUDs and DikuMUDs—it is possible for a wizard to become invisible (sometimes called "dark"), which enables him or her to spy on players.
- **Disciplining players**—Some players may abuse the system, cheat, harass other players, or just break the rules. Wizards are responsible for ruling on these complaints and taking whatever disciplinary action is necessary. Some reprimands include deleting the offending player, suspending his or her account for a period of time, banishing his or her site (no one from that site on the Internet can connect), and on combat MUDs, taking away experience points or gold.

- **Player problem resolution**—Bugs or other freak things often kill players, take their money, or otherwise impede them. And when this happens they usually complain to a wizard, because gaining levels and making money takes a lot of time. Wizards can, at their discretion, compensate players for lost experience points, gold, or items. Note that not all wizards can do this, however.

As you can see, some of these powers are quite enormous. The power to ultimately destroy anything that exists in the world, the power to read others' virtual thoughts (snoop), the power to create new space and creatures—these powers are the reason why the highest level wizards are often called gods. And in the MUD world, these powers are god-like. These powers do not come without responsibility, however.

On most MOOs, MUSHes, and MUCKs, there are only two levels: normal players and wizards. Occasionally you will see a reference to a god—that player generally has no real power as far as commands go, it's just a title. It denotes the lead wizard of the group, the head wizard, or the wizard with the final authority. While on LPMUDs and DikuMUDs there are normal players, several levels of wizards, and then the gods (and occasionally a high god who has the final authority). Note as well that LPMUD and DikuMUD gods may not have additional commands that other wizards do not have (although often they do), but their greater power lies in the authority on MUD issues. It's a distinction that has to be made, as MOO, MUSH, and MUCK wizards are closer to being equivalent to LPMUD gods, not LPMUD wizards.

Most MUDs have one person or a small group of people where the power resides—often called the *gods* on MUD. These players (the gods) usually are the ones who own the machine, or the account on which the MUD runs, or the people who have written a vast majority of the MUDs. This group in power, usually the gods, have the power to make, promote, demote, and delete wizards. They usually are the ones who set the policy and deal with wizards who get out of line or abuse their powers (which happens fairly often).

As you can guess by the references to promotion, demotion, and a god-level wizard, there often are multiple levels of wizards. Levels are created based on administrative needs and are granted based on performance, aptitude, and seniority (and politics, but we won't go into that yet). The lowest level is the *basic wizard* or *builder*. This wizard's responsibility is to create new regions on the MUD. This type of wizard does not have the authority to handle player problems or other issues that may arise.

Above this level usually is an administrative or elder wizard who can compensate and discipline players, oversee and approve lower level wizards' work, and handle various other administrative tasks. This level reports directly to the god-level wizards.

This system of wizard hierarchy is not as likely to hold true on MUDs where everyone can build. Every wizard has access to the same commands on MOOs, MUCKs, and MUSHes, but wizards (as they do on LPMUDs and DikuMUDs as well) still tend to stratify according to their "jobs"—for example: wizards who are responsible for building inspection/control, wizards who focus on dealing with player conflict and complaints, and wizards who mainly program the core of the MUD.

Some MUDs may have fifty different wizard levels while others may only have one level, or even one wizard (although that is not likely). You'll need to get a feel for how the wizard levels work on each individual MUD.

What is a "Wizard" or "God"?

Gods are those who own the database—the administrators. In most MUDs, Wizards are barely distinguishable from Gods—they're barely one step down from the God of the MUD. An LPMUD Wizard is a player who has won the game, and can now create new sections of the game. Wizards are very powerful, but they don't have the right to do to you whatever they want; they still must follow their own set of rules, or face the wrath of the Gods. Gods can do whatever they want to whomever they want whenever they want—it's their MUD. If you don't like how a God acts or lets his wizards act toward the players, your best recourse is to simply stop playing that MUD.

A more appropriate name for wizard would probably be janitor, because wizards tend to have to put up with responsibilities and difficulties (for free) that nobody else would be expected to handle. Remember, they're human beings on the other side of the wire. Respect them for their generosity.

This should give you a basic feel for some of the powers and responsibilities of being a wizard. The following section details programming and specific wizard commands for various different kinds of MUDs.

Do I Want to Be a Wizard?

Do you want to be a wizard? This is a personal decision. If you think it would be fun to create a world rather than to play in one, then you probably would enjoy being a wizard. Being a wizard is a lot of work, and it isn't all fun. MUDs often have a lot of politics at the wizard level. This can affect promotion, having your work approved (some MUDs require a higher level wizard to approve new creations before they are added to the MUD), or even cause you to be deleted. MUD politics can be really bizarre, but then so can player politics.

Personally, I have been a wizard on several MUDs and have enjoyed it. But it also is nice to play or just hang out, without any responsibilities. And when you are a wizard, you will be the target of many `tells` and `shouts` from players looking for a wizard to help them with a problem. Being a wizard has its rewards, but also its drawbacks.

I've enjoyed being an active wizard on one MUD while at the same time, playing on one or two others. Doing this enables you to enhance one world as a wizard and take a break and kill monsters as a player. Some MUDs even allow wizards to have a second player character that they can use to run around with and kill a few monsters (the player has a wizard character and a normal player character, but both are controlled by the same real person). A nice break from developing a world. Being a wizard on multiple MUDs is pretty difficult. You'll probably end up only really working on one.

Deciding whether you want to be a wizard is a tough decision. It certainly can be fun to remain a player, and if you are on a combat MUD and considering being a wizard, it probably means you are one of the most powerful players around. It's always a bummer to go from being one of the most powerful players to low man on the totem pole (new wizards are pretty lowly). It can take a long time to work your way up to a high level inside the wizard hierarchy. Creating worlds is not always easy.

If you decide you want to be a wizard, good luck. If not, enjoy playing MUDs—for many, simply playing is more fun than being a wizard. If you want to become a wizard, read on!

How Do I Become a Wizard?

Becoming a wizard is one of the things that varies most among different MUDs and different types of MUDs. On social MUDs (MOOs, MUSHes, and MUCKs) and DikuMUDs, the appointment of wizards and their equivalents often is very political. Because social MUDs often have no levels or hierarchy among players, there is no real firm basis for a system to create wizards. Players often just start talking to wizards and they get to know each other. And next time a new wizard is needed, the wizards pick someone they know.

Not all MUDs base their selection process on "an old boy network" system, picking friends or wizards from other MUDs to become the new wizards—some use other methods as well. Some MUDs look for existing players that seem competent, familiar with the MUD, and willing to devote the time necessary to be a good wizard. Others use a more familiar, but sometimes more difficult, system like that found in the real world, such as resumes and job interviews.

It seems, however, that most wizards on social MUDs got to be wizards because they were around when the MUD was started so they got in on the ground floor. Or, they were wizards on another MUD, and when a MUD needed wizards, they looked for people with experience. Becoming a wizard on a MUD can lead to opportunities as a wizard on other new and existing MUDs.

If you want to be a wizard, it might be a good idea to look at the MUD-related newsgroups (see these in the MUD directory, resources section). Many new MUDs will post that they are looking for wizards on the newsgroups. They often will ask questions about your MUD experience, such as, "Have you ever been a wizard and on what MUDs and systems and what kind of work have you done?" It's a good idea to treat these questions like a job application and promote your creativity (as a good person to make new areas on the MUD) and to mention any programming experience you may have in real life (LPMUDs, for example, work a lot like UNIX and use a programming language similar to C). These extra advantages can help get you in the door as a new wizard. And if you decide you don't like the MUD, you now have experience and it will be easier to get a wizard position on another MUD.

The Wizard Structure on a MUSH

MUSH The Dark Gift MUSH (`128.2.21.47 6250`) has three different levels of what we have been calling a wizard: staff, admin, and wizard.

- **Staff**—To become a staff member, watch for a message on the MUSH or in the Usenet newsgroups requesting more staff members. Then submit a resume and a letter explaining qualifications and why you want to be a staff member. Then the current staff, admin, and wizards vote on the applicants. The responsibilities of the staff members are to help players. This might mean creating objects for players (on this MUSH, players can `buy` objects) or judging. This MUSH has a combat system that is not built into the MUSH, so it must be administered manually by a judge (staff members don't have many responsibilities for the MUSH as a whole).
- **Admin**—The admins are a step above the staff members. They usually are chosen from staff members who are performing well. To become an admin, you would want to be seen online often and perform well, and then, when an opening is available at the admin level, you will have a chance of being promoted. Admins are in a gray area between staff and wizards. They have some real authority—they can make buildings and approve things on a higher level than staff—but still report to the wizards.
- **Wizard**—A wizard oversees his or her department(s) and ensures that everything runs smoothly. Wizards primarily come from the admin level or have been around since the MUSH started. They generally make sure everything is done properly and deal with the things that come up that no one else is equipped to deal with.

As you can see, there are many arbitrary decisions in the process of appointing wizards on this MUSH. The existing wizards decide who should join their ranks and who should be promoted. This is not an uncommon system, but it is not the only system. It is likely that you will find MUSHes with fairly different structures in the hierarchy of wizards. The preceding example is just a sample of what you might expect to find.

Becoming a Wizard on an LPMUD

LP MUD LPMUDs have a defined procedure for becoming a wizard. To achieve wizard status on an LPMUD, there are two standard requirements: a certain level, and the completion of a set of quests. The level requirement is fairly obvious—you have to work your character up to a certain level, often 20 or 30. When you reach this level, you then can qualify to become a wizard if you have completed the other requirements. Look on the MUD you play for the specific level. On some LPMUDs, you can not advance past the wizard level—for example, if you want to remain a player, you might have to remain at level 19 forever. On most LPMUDs, you can advance past the level required to become a wizard, and then you can attempt to advance to wizard status at any time.

The other common requirement for becoming a wizard is the completion of a set of quests. A *quest* is a special adventure that has a specific goal. For example, a common quest (one of the default quests built into LPMUD) is called the *orcslayer quest*. It involves finding the orc shaman, a monster with a special sword called an orcslayer. The orcslayer is a short sword that does extra damage to orcs. After you have found the orc shaman, killed him, and retrieved the sword, you must return it to Leo the Archwizard. Leo the Archwizard usually resides in the basement of the church. After you return the sword, you will have completed the quest.

Usually, when you complete a quest, you also will get bonus experience points for solving the puzzles involved in the quest. Most are more complicated than the orcslayer quest, which usually is one of the first quests that people solve. You probably will want to look at solving quests regardless of whether you plan to *wiz* (the act of becoming a wizard). Quests bring experience points and other benefits. Sometimes quests are required before you are allowed to join a special guild or other group. Quests also are occasionally required to advance to certain levels, although this is pretty rare.

Most LPMUDs list the quests that are on the MUD (and how many you need to complete to become a wizard), as in the following example:

```
Welcome to The Adventurers' Guild
You have to come here when you want to advance your level.
You can also buy points for a new level.
Commands:  cost, advance, spend, list (number).
There is an opening to the south, and some blue shimmering
light in the doorway.

      There are two obvious exits: north and south.
a book in a chain.
> list
You have 10 quests unsolved.
You must solve 10 of these.
```

This particular MUD has an unusually high number of quests. As the ranks of existing wizards swell, you will find that some MUDs make it much more difficult to become a wizard. On other MUDs, it may be much easier, as the MUD may be in need of new wizards. The following sample session shows details of individual quests:

```
> list 1
Retrieve the Orc slayer from the evil orc shaman, and give it to Leo.
> list 2
You must compete in the Trial of Champions!
A galley will take you to Trial Island where you must listen to
Felionus Moneybags. He will give you your instructions. You can acquire a
galley
by blowing the Horn of Resounding which is found at one of the Realms docks.
This is for levels 15 and over.
```

Or if you have completed them all, you can use the following:

```
> list
You have solved all quests!
```

As you can see, the MUD doesn't provide you a whole lot of information on solving the quests. You may want to consult with other players to find out the details of what you must do for the quest. Because many MUDs have harsh penalties for those helping someone on a quest, players might not be eager to help you, but they should at least be able to provide some guidance.

On many LPMUDs, after you achieve the required level and complete all the quests, you will be able to automatically become a wizard in training—you don't need another wizard's approval. The way it usually works is that you walk into your guild hall and advance your level as normal. An announcement will go out to the MUD that is similar to "A new wizard is born." Now you are a wizard!

NOTE

On many older LPMUDs, you will need to log off and log back onto the MUD to become imbued with your new wizard powers (for your wizard commands to work).

On some LPMUDs, however, it is not quite so easy to become a wizard. For security reasons, many LPMUDs have removed the capability to become a wizard automatically, and now require another wizard's approval. This prevents players from quickly becoming a wizard without anyone noticing and wreaking havoc on the MUD.

Some LPMUDs have additional complicated requirements. These may include that players or wizards (or both) vote on new wizards, which might require you to be more proactive in making friends, by giving newbies help or money, or working with others. At the very least, you will need to avoid making enemies. Some LPMUDs have tests or other requirements. These additional requirements vary widely among MUDs.

Summary

This chapter has provided information about some of the powers and responsibilities of a wizard. This might give you some idea of whether you want to try and take the leap and become a wizard. If you do, you will want to read the next section of this book, which discusses details on programming and navigation of the various different types of MUDs.

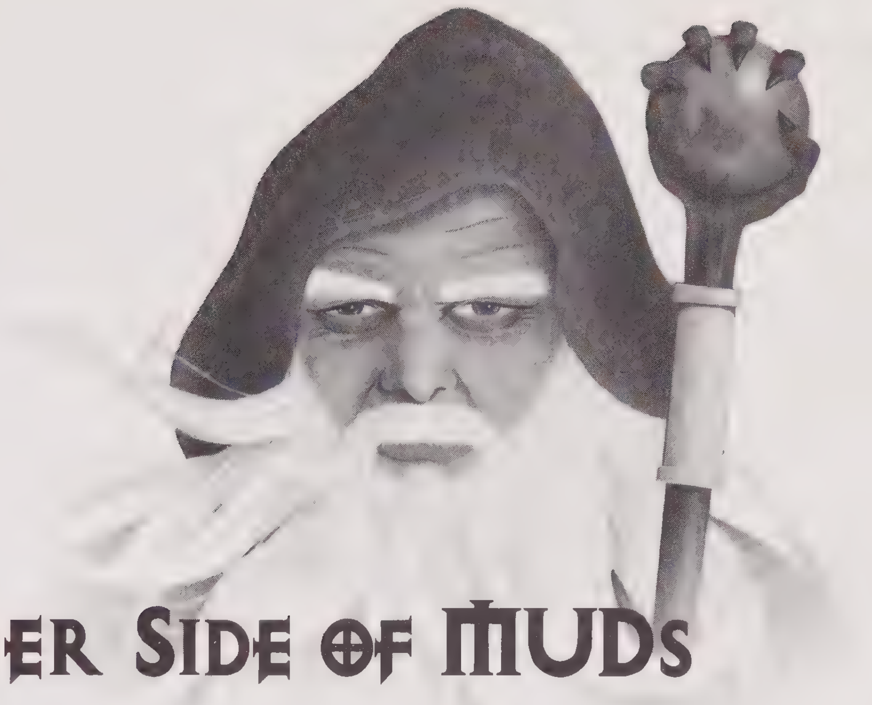
If you have decided that being a wizard is not right for you, you still might want to glance through the next section, especially the areas on the types of MUDs that you can choose to play. Understanding how the game works can give you an edge in playing.

III

PART

MUD Programming Guide

12
CHAPTER



The Other Side of MUDs

As we have discussed throughout this book, most types of MUDs are programmable. Some MUDs allow any player to create new objects using some custom interface, while other MUDs have advanced programming languages that are modeled from real world languages such as C and Forth. Unfortunately, in this area, MUDs tend to vary drastically. Programming an LPMUD is nothing like programming a MUSH.

MUD Servers

We have talked about using telnet and other clients to connect to a MUD. Up to this point, however, we haven't spent a lot of time talking about what a MUD is. When you connect to a MUD, you are connecting to a special piece of software called a *server* (perhaps you have heard of client-server software). MUDs work by having many simple clients connect to a server that processes all the requests and interaction of the clients.

Other multi-player games, such as DOOM, use another method, called *peer-to-peer*, to support several players. In a peer-to-peer game, no one computer handles all the interaction (as does the server on a MUD), but instead, all the computers send all user information to every other computer. This tends to create a lot of network traffic and generally is inefficient for a large number of players. Because of this, you will not find many peer-to-peer games that support more than eight players.

Now, after that brief diversion, let's talk about the MUD server. As you may have guessed, the distinctions among LPMUDs, DikuMUDs, MOOs, MUSHes, and MUCKs are the servers that run them.

As you play different MUDs that use the same server, you will find differences among them. This is because the majority of MUDs in existence are programmable. Not only is the source code of the server available for anyone with programming knowledge to change, but often the server itself has its own internal system—usually called the *MUDLib* or *core*—for creating the MUD world.

Different servers deal with programming in different ways. LPMUDs use a programming language called *LPC* (Lars Pensjl C), which resembles the popular C programming language (the C in LPC) and is very powerful. MUCKs use a language called *MUF—Multi-User Forth*. MOOs and MUSHes have their own systems that don't resemble existing programming languages for creating objects and developing the MUD world.

There are many types of MUD servers available on the Internet. The most popular ones are the five discussed in this book. Some of the servers discussed in this book have derivatives that closely resemble them, but are not necessarily the same. For example, the MudOS server is based on the LPMUD server, but has been developed along different lines than the current LPMUD server. You will find many similarities between LPMUD and MudOS, but you also will find many features that exist in one server, but not the other.

Another type of server, called *DGD* (Dworkin's Game Driver), on the surface looks to be similar to LPMUD and even incorporates the LPC language, but is not derived from the LPMUD server. DGD is a new server that is designed to be LPMUD-compatible but adds a new level of programming flexibility—in fact, MirrorMOO, running on a DGD server, emulates the MOO server.

The MUDLib

MUDLib is a term from the world of LPMUDs. On LPMUDs, the MUDLib is a set of program code written in LPC that implements central elements of the MUD, such as rooms, monsters, players, and the combat system. Because the MUDLib is written in LPC, it can be easily changed and altered without requiring that the MUD's server code be recompiled.

Because the set of programs that make up an LPMUD's MUDLib often interact with each other (for example, the monster component and the player piece interface with and use the combat system), they are *grouped*, which is why the MUDLib is considered a whole.

And, not surprisingly, several different MUDLibs develop and run different LPMUDs, or modify them or, in some cases, write them from scratch.

The uses of different MUDLibs contributes to the inconsistencies among different LPMUDs. MOOs also have something similar to a MUDLib that is called a *core*. The core is what makes MOOs different, like the many differences between LambdaMOO and Jay's House MOO.

Summary

The remaining chapters in this book cover three very different kinds of MUDs. They deal with the specifics of the MUD server. LPMUD is a very specific MUD server, which is why it has a different set of commands. DikuMUD, MOO, MUSH, and MUCK also are different varieties of servers.

You will want to read only the chapters in this section that relate to the MUD servers on which you want to work (or play).

Don't just put the book down now, thinking the rest of it is just programming. Be sure to glance at Appendix A, which provides a list of MUDs and other online MUD resources. If you don't read any of the programming chapters now, you likely will find them useful in the future. Once you become a wizard, or if you decide to starting creating objects on a MOO, MUCK, or MUSH, you will find the chapters on programming very useful.

13

CHAPTER

Essentials of LPC Programming (on LPMUDs)

Apprentice, wizard, and elder wizard—all are classic LPMUD titles for individuals who belong to a select group within a particular MUD. While the means by which one gains this status, as well as the title that goes with the status, may vary from MUD to MUD, the basic underlying concept remains the same: you are expected to contribute something to the game for the pleasure of both those who play the game and those who make it possible.

The duties of an apprentice usually involve two things: 1) learning his or her new place within the game, and 2) producing a Castle for players to roam. So where does one start?

Forces that are invisible to the eyes of players are set in motion the moment a player becomes an apprentice. On the hard disk where the MUD's data is stored, a subdirectory is created for the new apprentice. In addition, the new apprentice gains some commands that players do not have. On older MUDs, the apprentice may have to log out and then log back in to receive these benefits, while on newer MUDs, he or she may simply have to type `su` or do nothing at all.

NOTE Any time I indicate that one should enter *something*, whatever is in **`bold monospace`** may be followed by pressing the Return or Enter key (depending on which key the keyboard has).

Apprentices usually must report to a mentor (often called a sponsor) who has either been designated as such, or who has willingly accepted an apprentice under his or her wing. This sponsor, if there is one, is the person to whom you should address questions or bring problems, should either arise.

Not all LPMUDs allocate sponsors for apprentices. Instead, such MUDs usually group apprentices and low-level wizards into some kind of hierarchy, with a (hopefully) capable figurehead at key points within it. If this is the case, questions or problems should be brought to other wizards within or near the same hierarchical rank that he or she holds. If no one within those ranks can adequately provide answers, the next level of the hierarchy should be queried, until a solution is found.

At the very beginning an apprentice should not worry about his or her directory and should concentrate instead on his or her new set of commands. This is important because the commands are they keys to moving within the MUD's environment.

New Commands

The commands gained by new apprentices are fairly standard across all LPMUDs as they are the means by which apprentices make their contribution to the game. Some MUDs may provide more commands than are discussed in this section. Any unaddressed commands you gain are extra embellishments that some thoughtful person saw fit to add to make the lives of all easier on the MUD on which he or she codes.

As you know from having experienced the player's perspective of a MUD, the commands one issues to a MUD sometimes expect arguments or parameters. Also, as you have already seen, the *order* of arguments is important. In addition, the *type* of argument is just as important. Consider the following command:

```
tell Tarod Weren't you a God somewhere?
```

The command `tell` expects two arguments. The first argument is expected to be the player's name to which you want to `tell` (`Tarod`) while the second argument is expected to be what you want to `tell` the player (`Weren't you a God somewhere?`).

Similarly, apprentice commands expect arguments. You need to be aware, however, that the *type* of arguments becomes more relevant for apprentices. Some arguments should be strings while others should be integers or objects.

So what's a string? Simply put, a *string* is a group of alphanumeric symbols, commonly called *characters*. What's an integer? An *integer* is a whole number. Integers can be negative, positive, or even zero. What's an object? This is perhaps the most difficult type to grasp, as an *object* can be anything in the game that has been given definition as an object. Examples of common objects are players, monsters, weapons, armor, treasure, and torches.

Now that you know what type of arguments can be expected by apprentice commands, it's time to find out what the common commands are, how many arguments they expect, the order in which the arguments should be given, and what type each argument should be.

Environment Commands

Apprentices are given a wider range of freedom than are players, when it comes to expressing themselves online. Certain commands that seem harmless can be put to good use by players to cheat at the game, thus, such commands are reserved for the coders, knowing that the coders no longer play the game and have no reason to abuse the freedom. Apprentices also have a lot more to keep track of (regarding a MUD), than players do. Some common commands that allow more freedom of expression enable apprentices to keep track of things relevant within a MUD, and enhance a coder's communication capabilities are covered here.

earmuffs

While some MUDs allow players to have earmuffs, most don't. `earmuffs` is a command that allows you to ignore shouts on a MUD. On some MUDs, `earmuffs` is either on or off. For this type of `earmuffs` command, a single string argument consisting of the word `on` or `off` should be supplied to turn `earmuffs` on or off, respectively. Most MUDs, however, have evolved beyond this and allow `earmuffs` to be set to a specific level whose integer value cannot exceed your level as an apprentice. When set to an integer value, `earmuffs` screen out all shouts from players whose level is less than the specified level.

To turn on this type of `earmuffs`, invoke the command with a single integer argument whose value is greater than `0` and less than or equal to your level. If you specify an integer argument whose value is greater than your level, the argument will default to equal your level. Turning off this type of `earmuffs` varies from MUD to MUD. Typically, the command is invoked with a single integer argument of `0` or `-1` to turn off `earmuffs`. More advanced MUDs enable you to turn off `earmuffs` by issuing a single string argument consisting of the word `off`. Invoking either type of `earmuffs` command with no argument will tell you the status of your `earmuffs` (for the first type of `earmuffs`, either `on` or `off`; for the second type of earmuffs, what level `earmuffs` is set to). The following examples are type1 `earmuffs`:

```
earmuffs

earmuffs on

earmuffs off
```

The following are type2 `earmuffs`:

```
earmuffs

earmuffs 25

earmuffs 0

earmuffs -1

earmuffs off
```

NOTE Concerning the preceding example of type2 `earmuffs`, I assume for the sake of example that the wizard level of the apprentice is 25.

echo

`echo` is a command akin to `emote`. `echo` expects a string argument that can be anything. When invoked, `echo` prints the specified argument to all players in the room with you. This command is useful in conjunction with emote, as it does *not* prepend your name to the argument like emote does. Following is an example:

```
echo The grass withers and dies around Wolvesbane's feet.
```

echoall

`echoall` is an expansion of the `echo` command. `echoall`, too, expects a string argument that can be anything. Where `echoall` differs from `echo` is in its output. Where `echo` prints to all players in the room with you, `echoall` prints to all players on the MUD! Your name, as with `echo`, is *not* prepended to the string. Many MUDs do not allow apprentices to have this command immediately, as it can be annoying if overused. Following is an example:

```
echoall A cold feeling of uneasiness settles about you.
```

localcmd

`localcmd` scans you, everything you have, and the room in which you can be found to determine all the commands you can invoke. The command takes no arguments and will, upon invocation, print a list of the commands you can use.

people

The `people` command prints a list of facts about every player, much as a `who` list does. It takes no arguments and outputs the following information (about each user) to your screen in a columnized form:

- The `IP/ADDRESS` from which a user is connected
- The user's name, level, and age

- Whether the user is idle
- The path of user's current room

The information is columnized from left to right, and is preceded by a header that yields the following:

- The total number of users
- How many of those users are active
- How many commands per second are being executed (averaged over the last 15 minutes)
- How many lines per second have compiled (averaged over the last 15 minutes)

The following is an example of typical output from the `people` command.

```
There are now 43 players(41 active). 10.46 cmds/s, 2.68 comp lines/s
     =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
     128.174.5.60    Tarod          10001   96 D    ~tarod/workroom
     140.253.3.40    Syren          700     88 D    room/root/board_room
     192.100.81.121  Sonja          150     36 D    room/root/board_room
     128.123.34.5    Bud            36      24 D    ~animal/light/light4
     138.202.30.25   Hangman        35      37 D    ~animal/light/light4
     156.151.3.4     Iceshadow      33      7  D    ~animal/houses/house
     131.114.11.33   Sturm          33      14 D    ~syren/castle/doom_c
     128.206.115.39  Ravana         33      18 D    guild/thief/thief_gu
     198.3.127.2     Librarius      30      8  D    ~quest/entrance/room
     129.63.152.2    Lure           28      11 D    guild/monk/rooms/mon
     35.8.2.61       Droopy         25      10 D    guild/thief/room/fen
     128.169.202.68  Carnivor       24      12 D    ~craysus/tower/goodr
     142.150.1.22    Aurora         23      6  D    ~ambar/kelamor/room9
     137.229.10.33   Milk           21      6  D    ~maeglin/minas_tirit
     128.95.136.13   Flavius        20      9 D     ~gor/castle/5
     198.3.127.1     Bobble         20      12 D    ~syren/castle/doom_c
     137.229.10.33   Krink          20      7  D    guild/monk/rooms/mon
     128.123.34.14   Belgarion      19      4  D    ~maelik/masyria/sour
     158.121.2.2     Basil          18      6  D    ~craysus/tower/goodr
     155.238.33.183  Carmen         18      9 D     guild/druid/druid_gu
     198.3.127.1     Fiz            18      9 D     ~craysus/tower/good1
     193.190.1.55    Marcus         17      5  D    ~jaymz/cave/cave3
     192.17.3.3      Estios         17      5  D    ~syren/castle/hfores
     140.247.79.71   Corman         17      2  D    ~ambar/kelamor/room7
     193.190.1.46    Gorby          16      4  D    ~gor/plains/dark
     155.238.16.8    Kelemvor       16      2  D    guild/monster/rooms/
     150.203.66.218  Kiri           16      2  D    ~jaymz/cave/cave3
     193.190.1.22    Carnaval       14      2  D    room/root/vill_road2
     193.190.1.63    Morgana        12      0  D    ~sensual/xtalcaves/R
     130.191.1.4     Jenga          12      13 h I  guild/monk/monk_guil
     129.63.152.2    Sutek          12      11 h    room/root/board_room
     129.186.148.21  Ged            12      2  D    ~quest/tyrsis/iw9
     142.150.1.22    Kiger          11      8  h I  guild/fighter/fighte
     140.142.63.4    Hardcore       11      10 h    ~heart/castle/guards
     130.253.1.13    Arianna        11      1  D    ~animal/houses/house
     192.100.81.126  Elghinn        11      18 h    ~sensual/xtalcaves/R
     128.8.70.8      Shmoove        10      1  D    ~taran/room/entrance
     138.77.37.37    Jaden          8       5  h    ~gor/hole/hall4
     165.113.1.40    Ondska         8       13 h    ~sensual/xtalcaves/R
     138.77.37.37    Bacho          8       10 h    ~gor/hole/hall4
```

```
128.169.202.68  Chira          7       5  h    ~hulk/newbie/ghost_e
36.8.0.62       Ezekiel        6       5  h    guild/monster/rooms/
150.216.1.239   Tiny           6       6  h    guild/fighter/fighte
=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.
```

wiz

The `wiz` command is just like a `shout` command that shouts only to other wizards. It takes a string argument that is anything you would care to say to all the wizards who are logged on. Following is an example:

```
wiz Has anyone seen my sponsor?
```

wizlist

The `wizlist` command has been around for a long time on LPMUDs. It was added to enable MUD Gods to keep track of which castles were popular with players and to help the Gods find out why. Originally, being a wizard was a game in itself. Wizards earned points for use of their area and were supposed to be awarded rank based on these points. However, this mode of thinking in LPMUDs is long outdated.

A wizard gets one point for every command he or she has defined when it is used by a player. When a wizard uses commands defined by his or her objects, he or she gets points for that, as well. The score decays by one percent every `reset`, and is saved and restored from a file when the game is rebooted. If there are many wizards on a MUD, not all `wizlist` data is printed. If a wizard does not appear on the list, you can invoke `wizlist` with a single string argument that should be the name of the player whose score you want to see.

`wizlist` output is columnized to provide the following information:

- Score of a wizard's castle
- Score percentage of the total
- Rank sorted by castle score
- Total number of evaluated nodes
- Total number of heartbeats
- Total number of indices used in arrays by the wizard's files.

The following is and example of typical output from the `wizlist` command.

```
Wizard          Points %  Rank  Eval'd Nodes  HrtBts  Indices
=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.
sboe               908  0% (23) [2010k, 3412]     53674    479
dyv               1122  0% (22) [ 668k, 2438]    231343    226
tarod             1442  0% (21) [ 568k, 1004]    555671    200
aviar             3522  0% (20) [1472k, 4467]    161518    232
megadeth          3524  0% (19) [2488k, 3797]   2028522   1918
quest             5141  0% (15) [2807k,22253]    715688   1068
ted               6728  0% (14) [3037k, 3261]    356952   1251
barbie            7633  0% (13) [1735k, 3540]  -2091785   1193
```

```
heart          9502  0% (12) [3598k,11281]    1355310    460
jaymz         13877  0% (11) [7596k,12793]    2378966   1312
huma          14029  0% (10) [1739k,    0]      -1653     39
syren         14042  0% (9)  [12456k,28344]   7447407   3619
craysus       14601  1% (8)  [6841k,10636]    2834291   1448
omega         15373  1% (7)  [8305k,23106]    5963098   2146
taran         20688  1% (6)  [6343k,12998]     797487    465
maelik        26909  1% (5)  [19313k,27846]  11965557  45165
gor           33672  2% (4)  [24908k,99072]   2797208   4478
ambar         41764  2% (3)  [20243k,62073]  12942656   3805
maeglin       47374  3% (2)  [9713k,13365]    8831847   2969
animal      1113832 78% (1)  [660309k,14933] -1180178  36129
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-==-=-=-=
Totals:     1410040     (64)
```

Movement Commands

Apprentices and their ilk don't walk around very often in MUDs—they have better things to do than spend 30 seconds walking to meet someone, get somewhere, or acquire something. There are three commands typically provided in order to make a coder's life easier.

goto

`goto` is a command that takes one argument and expects that argument to be a string. The string usually can be either a player's name or the full path of a room. The first form of `goto`, which uses a player's name, is the more commonly used, as in the following examples:

```
goto shadowlor
```

```
goto /room/church
```

The first example will, if Shadowlor is logged in, move you to the room in which Shadowlor can be found. If Shadowlor is not logged in, `goto` will tell you that it could not find him. The second example will, if /room/church exists, move you directly to it.

This movement has the effect of teleporting. You will cease to be where you previously were and reappear in the new location you specified, assuming all goes well.

trans

`trans` is a command that takes one argument and expects that argument to be a string. The string must be a player's name, as in the following example:

```
trans babrius
```

If Babrius is logged in, the preceding example would cause Babrius to be transported to your current location. If the game cannot locate Babrius, then nothing will happen and `trans` will notify you that it could not find Babrius.

This movement has the effect of teleporting. Babrius will cease to be where he previously was and will reappear in your current location, assuming all goes well.

home

`home` is a command that takes no arguments. Simply type it and press Enter, as in the following example:

```
home
```

Invoking the `home` command moves you to your workroom. You must have a file called `workroom.c` in your personal directory for this to work properly.

File System Commands

File system commands are perhaps the most important commands for apprentices. While movement commands save you time, file system commands are those commands that enable you to move around the MUD's subdirectories and look directly that what is contained therein.

cat

`cat` is a command that takes one argument and expects that argument to be a string. The string can be either the name of a file or the full path of a file, as in the following examples:

```
cat workroom.c

cat /players/bleys/workroom.c
```

If, as in the first example, only a file's name is given, the file will be searched for in the current directory and, if found, the contents of the file will be printed to the screen. If, as per the second example, a full path is specified, `cat` will attempt to print the contents of the file specified. If the file does not exist or you do not have permission to read the file, `cat` will inform you that it was not able to complete its task.

NOTE

If the file is large, characters beyond a certain point generally will be lost or the file will be truncated prior to this break point. This has to do with the way the game sends the file's contents to the screen. As a result, `cat` is not a good way to view files unless the files are small.

cd

The `cd` command stands for `change directory`. It takes a single string argument, in one of several forms, and usually defaults to changing to your own subdirectory if no argument is supplied, as in the following examples:

```
cd /log

cd OLD

cd ..

cd ~
```

The first example changes the directory with you are working to `/log`. The second example searches the directory in which you're working (`/log` if you do it immediately after the first example) for a subdirectory called `OLD` and, if found, makes it become your current working directory. The third example moves you one directory up in the directory chain (meaning your directory goes from `/log/OLD` to `/log` again if you are doing these examples sequentially). The fourth example takes you to your personal directory (known as a *home directory*) exactly as if you had typed `cd` without an argument.

The following uses the preceding examples.

```
> cd /log
cwd: /log/
> cd OLD
cwd: /log/OLD/
>  cd ..
cwd: /log/
>  cd ~
cwd: /players/bleys/
```

NOTE The ~ (tilde) usually is unsupported on older MUDs. Due to the existence of globally accepted standards within the MUD community, when the tilde is supported, it will always carry the meaning of "home directory" when used with file system commands.

cp

`cp` is the equivalent of the DOS `COPY` command and the UNIX `cp` command. It expects two arguments that are both strings. The strings can either be file names or full pathnames of files. If the tilde is supported, you can use it in either the first argument, the second argument, or both arguments. If the first argument supplied (the file you want to copy) cannot be located, or you do not have permission to read it, `cp` will inform you that it could not find what you told it to copy and, as a result, will do nothing. `cp` usually does not allow you to overwrite a file that already exists. On some MUDs, however, where `cp` has been given the capability to overwrite an existing file, the command will prompt you, asking if you want to overwrite a file or not. Just as `cp` does not allow you copy a file you are not authorized to read, it does not permit you to copy a file to a directory to which you do not have write access. The following are examples of using `cp`.

```
cp /players/bleys/workroom.c /players/bleys/test.c

cp ~/test.c ~/test1.c

cp test1.c /players/bleys/test2.c

cp test2.c test3.c
```

The first example illustrates the use of cp with full paths given as arguments. The second example illustrates the use of the tilde in both arguments, while the third example illustrates the use of a filename as the first argument and a path as the second argument. The fourth example illustrates the use of filenames as both arguments.

The following uses the preceding cp examples. I cd'd (with no argument) first to get into my home directory for this example.

```
> cd
cwd: /players/bleys/
> cp /players/bleys/workroom.c /players/bleys/test.c
cp: /players/bleys/workroom.c copied to /players/bleys/test.c
> cp ~/test.c ~/test1.c
cp: /players/bleys/test.c copied to /players/bleys/test1.c
> cp test1.c /players/bleys/test2.c
cp: test1.c copied to /players/bleys/test2.c
> cp test2.c test3.c
cp: test2.c copied to test3.c
```

NOTE

When using full paths, you can leave off the filename (the last part in a full path) in the second argument. If you do this, cp checks to make sure that the path you give exists and is a directory and, if it is, copies the file specified by the first argument into the directory indicated by the second argument. You can do this when using a filename as the second argument, as well. Simply specify a subdirectory (whose location is within your current working directory) rather than a filename for the second argument. The cp command checks to make sure the second argument exists and is a directory and then copies the file into the directory if both checks return true. If supported, you can use the tilde for this application of cp, as well.

The following are examples of how to use cp to copy files into a subdirectory without specifying the destination filename:

```
cp workroom.c /players/bleys/NEWDIR
```

```
cp test1.c NEWDIR
```

Assuming that you are in your home directory and a subdirectory, NEWDIR, exists within it, the first example will copy workroom.c into NEWDIR, retaining the name of the first argument. The second example will copy test1.c into NEWDIR, as well. Again, the filename will be retained.

The following uses the preceding examples. I cd'd (with no argument) first to get into my home directory for this screen shot. I then invoked mkdir (covered later) to make a directory.

```
> cd
cwd: /players/bleys/
> mkdir NEWDIR
mkdir: created directory 'NEWDIR'
> cp workroom.c /players/bleys/NEWDIR
cp: workroom.c copied to players/bleys/NEWDIR/workroom.c
```

```
> cp test1.c NEWDIR
cp: test1.c copied to NEWDIR/test1.c
```

ls

To get a list of files within a directory, you invoke the `ls` command. It accepts a single string argument that should be the directory path you want to list and usually defaults to listing the contents of the current working directory if no argument is supplied. On MUDs where the tilde is supported, it can be applied. Most LPMUD `ls` commands display the total number of blocks the files in a directory use on the MUD's hard disk. In addition, the number of blocks each file uses up is displayed next to the file's name. A block is 1024 bytes, which is the equivalent to 1KB of space used. The following are examples using `ls`.

```
ls
```

```
ls ~/NEWDIR
```

The first example yields a list of the contents of your current working directory. The second example lists the contents of `NEWDIR`, a subdirectory within your home directory.

The following uses the preceding examples. I `cd`'d (with no argument) first to get into my home directory.

```
> cd
cwd: /players/bleys/
> ls
/players/bleys:
Total 56
   1 NEWDIR/   11 test1.c   11 test3.c
  11 test.c    11 test2.c   11 workroom.c

> ls ~/NEWDIR
/players/bleys/NEWDIR:
Total 22
  11 test1.c   11 workroom.c
```

mkdir

The `mkdir` command creates a new directory. It expects a single string argument that can be either a name for the new directory or the full path of the new directory. If the new directory you specify already exists as a file or a directory, `mkdir` will be unable to complete its task and will inform you of the problem. In addition, you cannot use `mkdir` to create directories in places where you do not have permission to write, because creating a directory requires writing to the hard disk. On LPMUDs where the tilde is supported, you can apply the tilde when using a path for the `mkdir` command. The following are examples of using `mkdir`.

```
mkdir NEWDIR
```

```
mkdir NEWDIR/testing
```

The first example makes a new directory, `NEWDIR`, in your current working directory. The second example creates a new directory, `testing`, in `NEWDIR`.

more

The more command is an interactive extension of the cat command. As more is very MUD-specific, the way it works varies to a great degree. It is used primarily for reading files without editing them. more expects a single string argument that usually can be a filename or the full path of a file. You can use the tilde where applicable. more prints a full screen of text from the specified file to your screen and waits for input. The input varies from MUD to MUD so only the basics are covered, which consist of pressing Enter or entering q and then pressing Enter. When a screen of information is displayed, more pauses and waits for you to tell it what to do next. Pressing Enter prints the next screen of text from the file. Typing **q** and pressing Enter discontinues use of more and returns you to your normal prompt.

mv

mv is the equivalent of the DOS MOVE command and the UNIX mv command. It expects two arguments that are both strings. The strings can either be filenames or full pathnames of files. If the tilde is supported, you can use it in either the first argument, the second argument, or both arguments. If the first argument supplied (the file you want to move) cannot be located or you do not have permission to read it, mv will inform you that it could not find what you told it to move and, as a result, will do nothing. mv usually does not allow you to overwrite a file that already exists. On some MUDs, where mv has been given the capability to overwrite an existing file, the command will prompt you, asking whether you want to overwrite a file. Just as mv does not allow you to move a file that you are not authorized to read, it does not permit you to move a file to a directory to which you do not have write access. You can use the tilde where it is supported, and you can use it when specifying paths, if applicable.

When you use mv, unlike cp, no copy of the file is left behind. Thus, you can use mv to rename a file in a directory—think of it as moving a file from one name to another. Its most common use, however, is in moving a file from one directory to another. Following are some examples of mv:

```
mv /players/bleys/test.c /players/bleys/mvtest.c

mv ~/test1.c ~/mvtest1.c

mv test2.c /players/bleys/mvtest2.c

mv test3.c mvtest3.c
```

The first example illustrates the use of mv with full paths given as arguments. The second example illustrates the use of the tilde in both arguments, while the third example illustrates the use of a filename as the first argument and a path as the second argument. The fourth argument illustrates the use of filenames as both arguments.

The following shows a screen of the preceding examples. I cd'd (with no argument) first to get into my home directory for this screen shot.

```
> cd
cwd: /players/bleys/
> mv /players/bleys/test.c /players/bleys/mvtest.c
mv: /players/bleys/test.c moved to /players/bleys/mvtest.c
> mv ~/test1.c ~/mvtest1.c
mv: /players/bleys/test1.c moved to /players/bleys/mvtest1.c
> mv test2.c /players/bleys/mvtest2.c
mv: test2.c moved to /players/bleys/mvtest2.c
> mv test3.c mvtest3.c
mv: test3.c moved to mvtest3.c
```

NOTE When using full paths, the filename (the last part in a full path) can be left off in the second argument. If this is done, `mv` checks to make sure that the given path exists and is a directory and, if it is, moves the file specified by the first argument into the directory indicated by the second argument. This can be done when using a filename as the second argument, as well. Simply specify a subdirectory (whose location is within your current working directory) instead of a filename for the second argument. The `mv` command will check to make sure the second argument exists and is a directory and will move the file into the directory if both checks return true. If supported, the tilde can be used for this application of `mv`, as well. On some MUDs, if `mv` is told to move a file to a directory that does not exist, you will be prompted to indicate if you want the directory created.

The following are examples of how to use `mv` to move files into a subdirectory without specifying the destination filename.

```
mv mvtest.c /players/bleys/NEWDIR
```

```
mv mvtest1.c NEWDIR
```

Assuming that you are in your home directory and a subdirectory, `NEWDIR`, exists within it, the first example moves `mvtest.c` into `NEWDIR`, retaining the name of the first argument. The second example moves `test1.c` into `NEWDIR`, as well. Again, the filename will be retained.

The following is a screen of the preceding examples. I `cd`'d (with no argument) first to get into my home directory for this screen shot. I invoked `mkdir` in a previous example to make a directory `NEWDIR`, which already exists, as a result.

```
> cd
cwd: /players/bleys/
> mv mvtest.c /players/bleys/NEWDIR
mv: mvtest.c moved to players/bleys/NEWDIR/mvtest.c
> mv mvtest1.c NEWDIR
mv: mvtest1.c moved to NEWDIR/mvtest1.c
```

On some MUDs, you can use the `mv` command to move one directory name to another (that is, rename a directory). You accomplish this the same way you move a file.

pwd

The `pwd` command stands for `print working directory`. `pwd` takes no arguments and simply displays your current working directory, when invoked. This command is useful for times when you manage to forget which directory you are working in, perhaps because the phone rang, or you were simply skimming rapidly through directories.

rm

`rm` is the DOS equivalent of the `DEL` command and the UNIX `rm` command. You use `rm` to remove files. It expects a single string argument that can be either a filename or a full path. If the tilde is supported and is applicable, you can use it. When invoked with a filename for an argument, `rm` searches for the file in the current working directory and removes it, provided that it exists and you have permission to write to it. If a path is specified as an argument, `rm` scans the specified path for the file to remove and removes it if, again, it exists and you have the appropriate permission. Following are examples of the `rm` command.

```
rm mvtest2.c

rm ~/mvtest3.c
```

The first example removes `mvtest2.c` from your current working directory if it exists. The second example removes `mvtest3.c` from your home directory.

rmdir

The `rmdir` command is `mkdir`'s counterpart. You use it to remove a directory. It expects a single string argument that can be either the name of the new directory or the full path of the directory. If the directory you specify does not exist or isn't a directory (that is, it's a file) `rmdir` will be unable to complete its task and will inform you of the problem. In addition, you cannot use `rmdir` to remove directories you do not have permission to write to because removing a directory requires writing to the hard disk. On LPMUDs where the tilde is supported, you can apply it when using a path for the `rmdir` command. Finally, `rmdir` will not remove a directory that is not empty. Before you can `rmdir` a directory, you must `rm` or `mv` all the files within it to a new location. Following are examples of the `rmdir` command.

```
rmdir NEWDIR

rmdir NEWDIR/testing
```

The first example removes the directory, `NEWDIR`, in your current working directory, if `NEWDIR` exists. The second example removes the directory, `testing`, in `NEWDIR`.

tail

The `tail` command is `cat`'s counterpart. `Tail` takes one argument and expects it to be a string. The string can be either the name of a file or the full path of a file. When invoked, `tail` prints the last 20 lines of the specified file. Following are examples of the `tail` command.

```
tail workroom.c
```

```
tail /players/bleys/workroom.c
```

If, as in the first example, only a file's name is given, the file will be searched for in the current directory and, if found, the last 20 lines of the file will be printed to the screen. If, as per the second example, a full path is specified, `tail` will attempt to print last 20 lines of the file specified. If the file does not exist or you do not have permission to read the file, `tail` will inform you that it was not able to complete its task.

NOTE How many lines `tail` will print to your screen is based solely on the discretion of the maintainers of a MUD. On some MUDs `tail` will print more than 20 lines, on others, less. However, the number of lines that `tail` prints can be expected to constant for a particular MUD. Twenty lines is merely the typical default.

Object Manipulation Commands

A number of commands are provided to enable you to manipulate objects. While files, manipulated by file system commands, are concrete things that exist on a hard disk, objects are not. Objects exist in the computer's memory, unless they have been swapped to hard disk to free up space for more objects. Thus, in order to manipulate objects, a special command set must exist. These are the commands that allow you to `load` an object from a file, copy it, update it, and even destroy it.

clone

`clone` expects a single string argument that can be a filename or a full path. The use of the tilde is permissible where applicable, if supported. The file to be cloned MUST end in `.c` for `clone` to work properly. If the object defined by the specified file is not currently loaded, on most MUDs, it will be loaded automatically upon invoking the `clone` command. `clone` ensures that the specified file is loaded into memory, after which a new copy of the object will be created, reset, and moved to either your inventory or to the room you are in, depending on the value returned by the function `get()`. Following is an example of the `clone` command.

```
clone /obj/torch.c
```

The preceding example clones the object defined by the file /obj/torch.c and moves it into your inventory. If you clone something that cannot be picked up, such as a bulletin board, it will be moved to the room in which you are located, rather than your personal inventory.

NOTE Rooms and castles should never be cloned; they should always exist singly to avoid wasting the MUD's finite memory.

dest

dest, short for "destruct," expects a single string argument that is an object's name. The object must be located either on your person or in the room with you for it to be destroyed. If the object to be destroyed is a cloned object, the memory-resident copy will continue to remain loaded. If, however, the object is a singular object (an uncloned object, such as a castle), then all data about the object will be discarded upon successful destruction of the object. An update of a singular object will have the same effect as desting it. Following are examples of the dest command.

```
dest torch

dest /obj/torch#8026
```

NOTE The dest command on many MUDs will also accept a single string argument that is a full path without the .c extension of the filename, to which an object number is appended. The need for this arises only when an object that has no name (due to a bug in the object's code, perhaps) must be destroyed. The object number of the object you want to dest must be known. Ask your sponsor or a wizard of higher level than you for information regarding this, should the need arise.

load

The load command expects a single string argument that can be a filename or a full path. When supported, the tilde may be used, if appropriate. The file to be loaded *must* end in .c for load to work properly. If the object defined by the specified file is already loaded into memory, nothing will happen. If not, the object is then loaded as requested, and reset() is called in the object. Following is an example of the load command:

```
load ~/workroom.c
```

update

The `update` command expects a single string argument that can be either a filename or a full path. The tilde may be used where appropriate, if supported. `update` destroys (as per `dest`) the memory-resident image of the loaded copy of the specified file and replaces it with a brand-new image loaded (as per `load`) from the specified file. If changes have been made to the specified file and it has since been updated, all *new* clones will behave according to the new memory-resident image, while clones made prior to invocation of the `update` command will continue to behave as per the code loaded into the previous memory-resident image. Following is an example of the `update` command:

```
update workroom.c
```

NOTE Updating a singularly existing object (like a castle or a room) has the effect of destroying and reloading it. Older MUDs also destroy *anything* (including players, wizards, items, and so on) that happens to be within the room when updated. More modern MUDs, however, will transport everything from the room before proceeding with the `dest` portion of the `update` procedure and, upon loading the new image, will transport everything back to the room.

Special Commands

There are two commands that need special attention, these being `ed` and `man`. While `ed` is supported on every LPMUD, `man` may not be. Both commands require knowledge of additional information. In `ed`'s case, there is a set of commands internal to `ed` itself. In the case of `man`, the argument that must be supplied is the name of a function. A list of functions when discussing `man` is provided for your use, as not all LPMUDs are kind enough to do so.

ed

The `ed` command is used to edit files. `ed` expects a single string argument that can be a filename or a full path. You may use the tilde if it is applicable and it is supported. Once invoked, `ed`'s subcommands must be used to perform any necessary modifications to the specified file. As `ed` is a line editor, simply entering a line's number at the `ed` prompt (usually a colon, :) will take you to that line. Entering the `=` displays that line's number. Entering `$` (called *string*), will take you to the last line of the file.

Table 13.1 lists all the common `ed` subcommands and their uses.

Table 13.1. Common ed subcommands.

Command	*Description*
`/`	Searches from line `1` to the last line for a pattern.
`?`	Searches backward from the last line to line `1` for a pattern.
`=`	Shows the current line's number.
`a`	Appends inputted text starting after the current line. When inputting text, entering a period, ., on a new line terminates input mode and returns you to the `ed` prompt.
`c`	Overwrites the current line with inputted text. When inputting text, entering a period, ., on a new line terminates input mode and returns you to the `ed` prompt.
`d`	Deletes specified line(s). If no lines are specified, `d` defaults to deleting the current line.
`h`	Displays help on `ed` subcommands.
`i`	Inserts inputted text starting before the current line. When inputting text, entering a period, ., on a new line terminates input mode and returns you to the `ed` prompt.
`I`	Indents the entire file's code.
`1,3j`	Joins specified lines together (in this case, lines 1 through 3).
`n`	Toggles on or off the line number display.
`1,$p`	Prints specified lines (in this case, lines 1 through the last line). If no range (such as 1,3 and so on) is specified, `p` defaults to printing the current line to your screen.
`q`	Quits `ed`. The file must be unmodified or have been saved to quit `ed` using this subcommand.
`Q`	Quits `ed` even if the file is file modified and unsaved.
`r fi`	Reads a file `fi` into the editor and appends its text at the end of the current file. You must specify the full path when indicating a file to read in and append.
`1,$s`	Searches and replaces through specified lines (in this case, lines 1 through the last line). You must specify what you want to look for and what you want to replace it with. The form of specification looks like `RangeBegin,RangeEnds/FindMe/ReplaceWith`. `1,$s/reset/create`, for example, replaces the first instance of `reset`, between the first and last lines, with `create`. If you wanted to replace *every* instance or `reset` with `create`, you would use `1,$s/reset/create/g` (the `/g` means global).
`x`	Saves current file and quits `ed`.
`z`	Displays 20 lines forward. If `z-` is specified, 20 lines backward will be displayed to your screen, instead.

Command	*Description*
z	Displays 40 lines forward. If z- is specified, 40 lines backward will be displayed to your screen, instead.

man

The man command stands for manual. It expects a single string argument that consists of the name of a function. When invoked, man will print to your screen information concerning the proper syntax and usage of the specified function. On some MUDs, man will yield a list of functions for which it can yield information.

Following is a list, arranged alphabetically, of functions normally supported on LPMUDs.

```
Commonly Available Manual Topics
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
add_action, add_xverb, all_inventory, allocate, atoi

break_string

call_other, call_out, call_out_info, caller, can_put_and_get,
capitalize, cat, catch, catch_tell, clear_bit, clone_object,  commands,
cp, create, create_Wizard, creator, crypt, ctime

deep_inventory, destruct, disable_commands, drop

enable_commands, environment, exec, exit, explode, extract

file_name, file_size, file_time, filter_objects, find_call_out,
find_living, find_object, find_player, first_inventory, function_exists

get, get_dir, get_localcmd, grab_file

heart_beat

id, implode, index, inherit_list, init, input_to, interactive, intp

living, log_file, long, lower_case, ls

map_array, member_array, mkdir, move_object

next_inventory, notify_fail

object_time, objectp

pointerp, present, previous_object, process_string, process_value

query_attack, query_auto_load, query_host_name, query_idle,
query_ip_name, query_ip_number, query_level, query_load_average,
query_name, query_prevent_shadow, query_snoop, query_verb

random, read_bytes, read_file, remove_call_out, rename, reset,
restore_object, rm, rmdir

save_object, say, set_bit, set_heart_beat, set_light, shadow,
short, shout, sizeof, slice_array, sscanf, stringp, strlen
```

```
tail, tell_object, tell_room, test_bit, this_interactive, this_object,
this_player, time

unique_array, users

write, write_bytes, write_file
```

Castle Creation

So far you have been exposed to commands that enable you to affect your environment more readily than players can. In addition, you now have the ability to move more freely and know more about the structure of the game than any player might. These tools give you the power to create the "magic" that players see when they wander through an area in search of fun.

You have been granted these things but not without responsibilities that go hand in hand with them. Your primary duty is contribute to the game. Apprentice contributions usually take the form of game land, known to old school MUDders and coders as castles. A castle is simply an area that a wizard has built and maintains. Every MUD differs on its castle policies. Restrictions are typically placed on how much experience can be given out for a monster, how much money and/or treasure the monster may have, what kind of quests you may build (if any), and so on.

NOTE

The restrictions just mentioned usually are dependent on the wills of the elders and gods of a MUD. Those specifics are not addressed in this book; however, note that it is *your* responsibility to learn, know, and adhere to the MUD's guidelines concerning areas/castles. More than likely, you have a sponsor to whom you can look for guidance. Ask your sponsor (if you don't have one, ask an elder wizard) where the guidelines for castle creation can be found, and then take the time to read them.

Originality is important. Many coders in days past have generated areas based on the content of fantasy or sci-fi books that they have read. While this is a fine contribution to the game, it does not demonstrate original, creative thinking. Consider a player who hops from MUD to MUD, sampling what each has to offer. Many times, he or she will go from one MUD to another and find similar castles as a result of some coder who did not have a mind of his or her own. I strongly suggest that once you have some idea of how the code works, you sit down with some graph paper and map out what your castle looks like, using one block for each room. Multiple levels are encouraged.

Realism plays an important part in creating an area, as well. How many times, as a player, did you walk into a room that had a well-written description that contained things you could not pay closer attention to or handle? It is recommended that any time you write a description, you make it possible to look at or manipulate every noun within said description. While this makes your coding more complicated, it also makes the game more vibrant and physical.

With these things in mind, it's time to learn how to build a castle. The best place to start is by learning which game version you are working with and where certain files are located within that game's MUDlib.

Game Version

Every LPMUD is different. From the moment you log on, you see differences in the way areas are laid out and their descriptions. Commands, too, differ from LPMUD to LPMUD. Beneath all of the aesthetics there is a programming platform called *LPC* that controls how things work. As this platform is developed by hobbyists and enthusiasts, it is always undergoing revision.

The program that keeps track of everything that occurs on an LPMUD is commonly called a "game driver." Without it, there is no game. This program has been around for years now and has undergone *many* changes. Some MUDs may be running older copies, while others are using the newest LPMUD (or compatible) driver available. While backward compatibility has, on the whole, been maintained, there are marked differences.

LPMUD versions 3.0 (or more recent) generally take up less memory and load faster than older versions. In addition, versions 3.0+ support more variable types and have more functions than versions 2.4.5 (and older). This does not mean that a MUD running on a 3.0+ driver is better than one running on a version 2.4.5 driver. It simply means that a MUD running on the 3.0+ driver is better at managing its resources than a MUD running on a 2.4.5 driver.

The transition point among code differences occurred when version 2.4.5 was still heavily used in the LPMUD world. Coders were accustomed to writing code using specific functions and, as a result, when version 3.0 was released, support for 2.4.5 compatibility was included within it. A 3.0+ driver can be compiled in what is known as *native* mode, which is not compatible with all 2.4.5 code, or it can be compiled in *compat* mode to allow old-style LPC code to be used.

You need to know which driver version the game you play uses, as this affects how you must write your code. If you don't already know the version of the game driver, or you're not sure, ask your sponsor or a high level wizard, to be sure. You also might ask the same individual to make you aware of any MUD-specific features in the driver that you may use.

The MUDLib and File Locations within It

MUDLib is short for *MUD Library*. The contents of the MUD's directory tree is called a library because much of what is contained therein can be viewed and used by wizards, as well as the fact that the game itself uses nearly all of it. Files within a MUDLib are akin to books on the shelves of a library.

Home Directories

Every apprentice and wizard has a home directory. This directory is where a coder builds his or her castle, as well as where any personal files (such as `workroom.c`) are kept. On LPMUDs running a 2.4.5 driver or a 3.0 driver in compat mode, *home* directories usually are located somewhere beneath `/players` (`/players/bleys`, for example). LPMUDs running a driver that is 3.0 native (or more recent) tend to locate wizard directories under `/usr`. Ultimately, where your home directory is located is dependent upon the will of the gods of a MUD. A simple `cd` command followed by the `pwd` command will show you where it is.

The *doc* Directory

A specific directory for documentation is generally maintained on each LPMUD. Such documentation usually is less important than game progress, so expect the documentation found within to be slightly, if not massively, out-of-date. This is not true on all MUDs, as some will assign an elder wizard or a small team of them to maintain the directory.

TIP

Information regarding wizard rules, wizard commands, castle restrictions, is typically kept in `/doc`.

The *log* Directory

Most LPMUDs maintain logs of events that transpire within the game. These logs are kept in order to be able to trace potential problems over a period of time. Things that are commonly logged include bugs reported, experience given, game errors, ideas reported, logins, logouts, player advancements, player deaths, reboots, shouts, treasure given, and typos reported. You can find things of this nature in `/log`.

The *object* Directory

Nearly every LPMUD keeps its base objects in `/obj`. Within `/obj` can be found the base files that make the game playable, such as `armor.c`, `container.c`, `living.c`, `monster.c`, `player.c`, `weapon.c`, and `treasure.c`.

The *open* Directory

The `/open` directory contains files and information to which all wizards, regardless of level, have access. If all wizards are able to write to this directory, as is the case on some LPMUDs, the information contained therein is generally unimportant, as anyone could alter it with ease. If, however, write access to `/open` is restricted; however, all wizards are permitted to read what is contained in the directory, and then expected to find relevant examples of code and tidbits of wizard humor within the directory.

The *players* Directory

Player files usually are stored in `/players`. Usually `/players` will contain subdirectories named for the letters of the alphabet, under which player filers whose name start with a specific letter will be stored. For security reasons, you usually are not permitted to read this directory.

The *secure* Directory

Nearly all LPMUDs maintain a directory that only high-level wizards may access. Often, the directory will be called `/secure` or `/closed`. Within this directory, you often will find fragile objects, that is, objects that can be easily broken unless they are handled carefully while coding. Certain crucial files, such as `master.c` (which enables the game to boot up), also are typically kept in a secure directory to keep them from being tampered with and keep ill-tempered wizards from perusing them for potential holes.

Comments

Now that you know where some of the essential files within the game are located, it is time to concentrate on the code contained within those files. A piece of code that works within the framework of LPC basically consists of four things: comments, variables, operators, and functions. *Comments* are placed in the code to make it clear to you (as a reminder for later, perhaps) what you did and, more importantly, to make it clear to others who may need to work on your code.

There are two ways to place a comment in your code. The first method, which works only on newer LPMUDs, is to preface your comment with `//`, as in the following:

```
//. This is an example of a comment.
```

Sometimes you may need to make comments that take up more than one line within a file. You may either place the `//` on each line of commentary, or you may use the second method of commenting, which works on all LPMUDs. The second way of commenting makes use of `/*` and `*/`. `/*` tells the compiler to recognize the beginning of a comment, while `*/` signals the compiler that the end of a comment has been reached.

A multiple line comment looks like the following:

```
/*
* This is an example of a comment
* that takes up more than one
* line within a file.
*/
```

Note that I indented my comment and carried the asterisk all the way to the end of my multiple-line comment. This helps to make the comment more visible and is strictly a matter of preferred form. I could just as easily have used the `*/` to indicate the end of my comment after the word `file`.

You can use this method of commenting for single-line comments, as well. A single line comment looks like the following:

```
/* This is a single line comment. */
```

Once you have mastered code and editing, you may find that while working on a file you inadvertently break something that used to work. A common method used to debug such a problem is to insert comments to change the newly-modified slices of code into something the compiler considers to be an explanation, and thus omits. If you accidentally break something while adding code, you should use this method to verify that what you modified caused the file to fail to load. When you have determined the location of the failing code, concentrate on fixing it.

Variables

Variables, another component of LPC code, are the means by which you store information temporarily. There are several types of variables, each of which is used to store a different type of information.

In order to use a variable, you first must declare the existence of that variable. Declaring a variable simply tells the compiler that it should reserve space for a particular type of data. Once you have declared the variable, you then can set it to whatever information you want to store.

Variables declared outside of a function are known as *global variables*, while those declared inside of a function are known as *local variables*.

Integer Variables

Integers, as you learned when exploring your new commands, are simply whole numbers ranging from negative infinity to positive infinity. Realistically, a computer cannot yet handle numbers near the limit of infinity. Normally, an integer can be a full 32 bits.

A declaration of an integer looks like the following:

```
int count;
```

The `int` portion of the preceding example tells the compiler to assign whatever follows it as integer variables. The second word, `count`, is the name of the variable. The semicolon at the end tells the compiler that you are done making an integer declaration.

Following are declarations of multiple integers:

```
int count1, count2;
```

This example works almost exactly like the previous example, except that more than one variable is declared. The comma is used to tell the compiler that you are done with one part—that it is okay to move on to the next part. Leaving the comma out and just using a space for separation will result in an error. You can use this method to declare three, four, or even more integer variables on the same line.

Status Variables

Status variables are used to keep track of things that can be toggled. A status variable, thus, is a Boolean. This means that the value of a status variable can either be `0` or `1`. Typically, `0` is considered `false` while `1` is considered `true`.

The declaration of a status variable looks like the following:

```
status stat;
```

The `status` portion of the preceding example tells the compiler that the variable type you are declaring is `status`. The `stat` portion is the variable's name and, as before, the semicolon ends your statement. As with integers, multiple status variables may be declared on one line.

String Variables

A *string*, as you learned from reading about your new commands, is simply a group of concatenated characters. Storage of strings, then, obviously is accomplished by using string variables.

The following is a declaration of a string variable:

```
string info;
```

Again, the `string` portion in the preceding example tells the compiler that the variable type you are declaring is a string. The `info` portion is the name of the variable. The semicolon ends your statement. As with integer and status variables, more than one string variable may be declared on one line, simply by separating them with commas. In fact, this is true of all variable types.

Object Variables

Object variables do not store objects, themselves, but rather, pointers to objects. It's almost as if an object variable were like a little man whose only job in life is to point at an object so that the compiler knows where to find it. Once you tell him what to point at, he continues to point at it until you either tell him to point somewhere else or go away. But, in order to tell him to point, you must first make him exist.

The following is a declaration of an object variable:

```
object ob;
```

As before, the first part of the statement, `object`, tells the compiler you're declaring an object variable. The name of the variable is `ob`, while the semicolon ends your statement.

Mixed Variables

Mixed variables are variables that store integers, strings, or objects. Declaration is as per the declaration of other variable types. The following is an example of a mixed variable:

```
mixed mixdvar;
```

Arrays

Arrays are an expansion of other variable types that can hold more than one piece of information. Arrays can be of type `int`, `status`, `string`, `object`, or even mixed. The information contained in an array can even be another array. (This is known as a *multidimensional array*.)

The following are examples of array declarations:

```
int *intarr;

status *statarr;

string *strngarr;

object *objarr;

mixed *mixdarr;
```

In the preceding examples, the first part of the statement is used to tell the compiler what type of variable you are declaring. The `*` indicates that the declaration is an array. The word attached to the `*` is the name you have given the variable, and as always, the semicolon ends your statement.

Arrays are stored by reference, thus, all assignments of whole arrays will just copy the address. An array will be deallocated when there is no longer a variable that points to it.

When a variable points to an array, information within the array can be accessed by indexing. Indices always begin with the first element, which is `0`, not `1`. Look at the following example:

```
arr[3]
```

The preceding example indexes into the array `arr` to get the fourth element, whose index is `3`.

The name of the array being indexed can be any regular expression or even a function call. Look at the following example:

```
get_name()[2]
```

The preceding example retrieves from the array the element indicated in brackets returned by the function `get_name()`. In this case, we have specified the element whose index is `2`. Thus, the third element of the array will be the result of the example.

You can construct arrays by generating a list inside `({` and `})`, as in the following example:

```
({ "this", "that", "the other", "none of the preceding" }).
```

The preceding example constructs an array whose size is `4`. The array will be initialized with elements whose values are the strings `this`, `that`, `the other`, and `none of the preceding`, respectively.

NOTE An array that is not initialized by using the array constructor should have its size allocated by using the `allocate()` function.

Type Modifiers

You can apply special *type modifiers* to the declaration of a variable. There are four modifiers available, as outlined in Table 13.2.

Table 13.2. Special type modifiers.

Type	*Effect*
`static`	The variable is not saved when `save_object()` is called on the object in which the variable is defined.
`private`	The variable cannot be accessed by an object that inherits the object in which the variable is defined.
`nomask`	The variable cannot be redefined by a shadow.
`public`	The variable can be accessed by an object that inherits the object in which the variable is defined. In addition, the variable may be redefined by a shadow.

If a type modifier is not stated in the declaration of a variable, it is understood to be of type `public`. You may use type modifiers singularly or use several at once. In the latter case, the type modifiers you opt to use must make sense. You might declare a variable as both `static` and `nomask` in order to disallow both saving by `save_object()` and redefinition (by a shadow), but you should not declare a variable as both public and private, as this is self-defeating. Variable declaration with type modifiers is outlined in the example that follows.

```
static nomask object ob;
```

The preceding example tells the compiler to acknowledge the existence of a an object variable `ob` that is both `static` and `nomask`. Had you wanted only to make the declaration `nomask` (to disallow redefinition by a shadow) it would have looked like the following:

```
nomask object ob;
```

The functions `save_object()` and `call_other()` have not yet been addressed. Also, the definition of a shadow has not yet been given. These things are covered in the section on functions under `save_object()`, `call_other()`, and `shadow()`, respectively.

Operators

Variables, as you know, are used to store information. LPC also provides you with ways to manipulate the information and expressions whose values you might store in variables. *Operators*, the third major component of LPC code, provide for such manipulation. Table 13.3 shows a list of the operators available in LPC in the order of precedence, lowest priority first.

Table 13.3. Operators available in LPC.

Operator	*Function*
`EXPR1 , EXPR2`	Evaluates `EXPR1` and then `EXPR2`. The returned value is the result of `EXPR2`. The returned value of `EXPR1` is discarded.
`VAR = EXPR`	Evaluates `EXPR` and assigns the value to `VAR`.
`VAR += EXPR`	Assigns the value of `EXPR+VAR` to `VAR`. An equivalent statement would be `VAR = VAR + EXPR`.
`VAR -= EXPR`	Assigns the value of `EXPR-VAR` to `VAR`. An equivalent statement would be `VAR = VAR - EXPR`.
`EXPR1 ¦¦ EXPR2`	The result is `true` if `EXPR1` or `EXPR2` is `true`. If `EXPR1` is `true`, `EXPR2` is not evaluated.
`EXPR1 && EXPR2`	The result is `true` if `EXPR1` and `EXPR` are both `true`. If `EXPR1` is `false`, `EXPR2` is not evaluated.
`EXPR1 == EXPR2`	Compares the values of `EXPR1` and `EXPR2` for equivalency. This operator can be applied to both strings and integers.
`EXPR1 != EXPR2`	Compares the values of `EXPR1` and `EXPR2` for unequivalency. This operator can be applied to both strings and integers.
`EXPR1 > EXPR2`	Compares the value of `EXPR1` to `EXPR2`. The result is `true` if the value of `EXPR1` is greater than that of `EXPR2`. This operator can be applied to both strings and integers.
`EXPR1 >= EXPR2`	Compares the value of `EXPR1` to `EXPR2`. The result is `true` if the value of `EXPR1` is greater than or equal to that of `EXPR2`. This operator can be applied to both strings and integers.
`EXPR1 < EXPR2`	Compares the value of `EXPR1` to `EXPR2`. The result is `true` if the value of `EXPR1` is less than that of `EXPR2`. This operator can be applied to both strings and integers.
`EXPR1 <= EXPR2`	Compares the value of `EXPR1` to `EXPR2`. The result is `true` if the value of `EXPR1` is less than or equal to that of `EXPR2`. This operator can be applied to both strings and integers.

Operators are not tricky. Using the preceding list of operators and their uses, you should be able to decipher mathematical operations and logical comparisons in code you `more` or `cat`. Eventually, after spending some time reading code to get a feel for it, you'll be ready to move on.

The following code sample combines some of what you already know about LPC into a series of logical operations that a game driver can comprehend and execute. You should read it slowly, thinking through the code as you go so that you understand what it is that the compiler is being told to do on each line. Comments are provided to help you.

```
/*
 * TESTCODE
 * This is a simple, sample piece of code. It illustrates the use
 * of operators as well as appropriate syntax. There will be things
 * contained within that you have not yet seen. For now, do not worry
 * about those things, as they will be explained soon. You should
 * note that variables, when declared, are uninitialized. All
 * variables, regardless of type, equal 0 when they have not yet been
 * initialized.
 */
int a, b, c;          // Declare 3 integer variables: a, b, and c
string tosay;         // Declare string variable: tosay
object ob;            // Declare object variable: ob
mixed any;            // Declare mixed variable: any

reset()               // Begin defining what happens when reset() called
{
  a = 1;              // set a equal to 1
  b = 2;              // set b equal to 2
  c = a+b;            // add a to b and set c equal to the result
  ob = this_object(); /*
                       * set ob equal to the value returned by the
                       * function this_object()
                       */

  if ( c == 3 )       // IF c is equal to 3 (it should be)
    {                   // THEN
    c++;                // increment c by one (to make it equal 4)
    }                   // END THEN
  else                // ELSE c isn't equal to 3 (this can't be true)
    {                   // SO THEN
    c = 4;              // set c equal to 4 (this should never happen)
    }                   // END THEN portion of else

  if ( c && c != a )  // IF c has a value (c!=0) AND it doesn't equal a
    {                   // THEN
    any = c+"\n";       // set any equal to c (4) plus a carriage return
    tosay = "C is "+any;// set tosay equal to the string "C is "+any
    }                   // END THEN portion
  else                // ELSE c must either be 0 or equal to a (not true)
    {                   // SO THEN
    any = "FAILED\n";   // set any equal the the string "FAILED\n"
    }                   // END THEN portion of else

  write(tosay);         // write the value of any to the screen
  destruct(ob);       // destroy ob (this object)
  return 0;             /*
                         * end this function by returning.  If no
                         * return statement is included, a function
                         * is assumed to return 0
                         */
}                       // End definition of reset() function
```

The preceding routine is about as simple as they come. It *should* work on all LPMUDs that are using 3.0 native (or more recent) drivers. If you're tinkering on an older MUD, then it might not work, as older MUDs don't usually support `//` comments. Try editing a file called `test.c`. From the `ed` prompt, use `a` to append. Then type in this program. You can leave out the comments if you want. When you finish typing it in, enter a period on a new line to stop appending. Then use `x` to save the file and exit `ed`. Your next step should be to `clone test.c`. Upon doing so, you should see `C is 4.` appear onscreen on a line by itself. At that point the object will self-destruct.

Syntax

Syntax is extremely important in LPC, as with all coding languages. If your syntax is incorrect, it will cause errors when the game driver tries to load a definition file. These errors usually are logged somewhere in `/log` for you.

Braces

Braces, `{` and `}`, are used to indicate the opening and closing of a function, a clause in an `if-else` statement, or the clause of a loop. The open brace, `{`, indicates the beginning of a clause, while the close brace, `}`, indicates the end of one. Look at the following example:

```
id(str) { return str == "An object"; }
```

In the preceding example, the braces denote the opening and closing of a function.

The next example illustrates the use of braces to indicate the clauses in an `if-else` statement.

```
if ( a == b )
       {
       a = a++;
       return a;
       }
     else
       {
       return a;
       }
```

In this example, braces denote the opening and closing of clauses in an `if-else` statement. What is contained within the braces is the `then` clause of all such statements.

This following and last example shows braces used to denote the body of a `for()` loop.

```
for( i = 0; i < 10; i++ )
{
write( "I equals: " + i + ".\n");
}
```

The body of a `while()` loop is donated exactly like that of a `for()` loop.

Semicolons

You use a semicolon (;) to indicate the end of a statement.

Parentheses

You use parentheses, `(` and `)`, to denote the arguments given to a function, as well as the order of operations within an expression, as in the following examples:

```
add_action("drop_thing", "drop");
```

```
int a, b;
    a = 1;
    b = 2;
    write( "a plus b equals " + (a + b) + ".\n" );
```

The first example illustrates the use of parentheses to denote the arguments to a function while the second example uses parentheses to ensure that `a` is added to `b` before the rest of the expression is evaluated.

Return Statements

Every function must end. You use the *return* statement to end a function by returning a value. If no return statement is found in the function, then the function is assumed to return a value of `0`. Because all variables have a value, you can return variables, as well. Any variable type, even arrays, may be returned. In addition, because all functions return something, you may return a function call!

Conditional returns (that is, a return that is dependent on the truth or falsehood of an `if-else` evaluation) are commonly used to end the evaluation within a function in a timely manner. When you write code, you should return where appropriate to avoid any unnecessary evaluation. This is good coding practice.

if-else

The `if()` statement in LPC is identical to that provided by the C coding language. An `if()` statement is used to test the conditions of an expression for truth or falsehood. An `if()` statement is followed by a body of code (that may or may not need to be placed within braces, depending on how much code follows the `if()`). This body of code is the `then` clause of an `if()` statement. The `then` clause is evaluated only if the `if()` statement is true. Look at the following example:

```
if ( a == b ) return a;
```

In the preceding example, the `if()` statement checks to see if `a` is equal to `b`. If this is `true`, then the body of code that makes up the `then` clause is evaluated; that is, the value of `a` is returned.

You can express `if()` statements as follows:

```
if (expression) statement;
```

```
if (expression) { statement; }
```

```
if (expression) statement;
```

The preceding three examples all mean exactly the same thing. The space between `if` and the first parenthesis is not necessary, but is commonly used to help make `if()` statements more visible in the code. You can put braces around `then` clauses that consist of only one statement for clarity, if you feel the need.

When more than one statement is necessary in the `then` clause of an `if()` statement, braces are *required* to embody the `then` clause, as in the following example:

```
if (expression)
    {
    statement;
    statement;
    statement;
    }
```

In this example, braces are *necessary*, as multiple statements are made in the then clause of the `if()` statement.

An `if()` statement may be followed by an `else` clause, as well. The `else` clause is evaluated only if the expression within the `if()` statement is `false`. Look at the following examples:

```
if (expression) statement;
else statement;
```

```
if (expression)
  {
  statement;
  statement;
  statement;
  }
else
  {
  statement;
  statement;
  }
```

The number of clauses of an `if()` statement is not explicitly limited. You may follow an `if()` statement with another `if()` statement in the `else` clause of the first `if()`, as in the following example:

```
if (expression0)
  {
  statement0;
  statement0;
  }
else if (expression1)
  {
  statement1;
  statement1;
  }
```

```
else if (expression2)
  {
  statement2;
  statement2;
  }
```

The statements inside the `then` clause of an `if()` statement may be `if()` statements, themselves. This is called *nesting* and is common. Look at the following example:

```
if (expression)
  {
  statement;
  statement;
  if (expressionA)
    {
    statementA;
    statementA;
    }
  }
else
  {
  statement;
  statement;
  }
```

By now, you have noted that in each example, I indent my braces two spaces starting on a new line, and have no statement on the same line that the braces are on. This makes for easier reading, as all clauses are indented, and thus, easier to see. You can place your braces and statements where you want, but try to make it easy to read your code in case others want to use it as an example or must work on it to help you fix it. Also note that you can nest `if()` statements in the `else` clause of an `if()`.

Loops

You use a *loop* whenever you need to perform the same operation a number of times. There are two types of loops: `for()` and `while()`. Use of the `for()` loop is preferred and should be used where possible when a loop is needed.

for()

The LPC `for()` loop is identical to that provided by the C coding language. As per `if()`,braces are not needed if only one statement makes up the body of code to be executed during the loop. Braces *are* required if more than one statement is to be executed throughout the loop. Look at the following examples:

```
for (expression0; expression1; expression2) statement;

for (expression1; expression2; expression3)
          {
          statement;
          statement;
          }
```

`expression1` is evaluated once prior to the execution of the loop.

`expression2` is evaluated at the beginning of each iteration of the loop.

The loop will be terminated if `expression2` evaluates to `0`. `expression3` is evaluated at the end of each loop iteration. Look at the following example:

```
int i;
for (i = 0; i < 10; i++)
  write("I == " + i + ",    10-I == "+ (10 - i) + "\n" );
```

The preceding example declares an integer variable `i`. Before the execution of the `for()` loop begins, `i` is set to `0`. The loop execution begins and the body of the loop writes `I == 0, 10-I == 10`. The variable `i` is then incremented by one and the loop is executed again. On this second pass, the output would be `I == 1, 10-I == 9`. The loop will terminate execution when `i` is equal to `11`.

Following is an example of the loop's output:

```
I == 0,   10-I == 10
I == 1,   10-I == 9
I == 2,   10-I == 8
I == 3,   10-I == 7
I == 4,   10-I == 6
I == 5,   10-I == 5
I == 6,   10-I == 4
I == 7,   10-I == 3
I == 8,   10-I == 2
I == 9,   10-I == 1
I == 10,   10-I == 0
```

A `break;` statement in the body of the loop terminates the loop. A `continue;` statement resumes the execution from the beginning of the loop after evaluating `expression3`. Both the `break;` and `continue;` statements should *always* be followed by a semicolon, as with any statement that is not followed by a clause or body of code.

while()

The LPC `while()` loop also is identical to that provided by the C coding language. The placement of braces to indicate the body of the loop is as per the placement of braces when using `for()`. The syntax of a `while()` loop is as follows:

```
while (expression) statement;
```

```
while (expression)
  {
  statement;
  statement;
  }
```

The statements inside the body of a `while()` loop are executed repeatedly until the expression evaluates to `0`. If the expression evaluates to `0` just prior to the execution of the loop, then the body of the loop will not be executed. As per `for()`, `break;` or `continue;` statements may be made within the body of the loop to stop or start its progress. Look at the following example:

```
int i;
while (i < 10)
  {
  write("I == " + i + ",   10-I == " + (10 - i) + "\n");
  i++;
  }
```

The output of the preceding example will look exactly like the output of the example previously shown in the section on `for()`.

In the previous example, had I failed to increment `i`, the result would have been an endless loop. LPC, however, is designed to permit only 50,000 evaluations at a time. Thus, the preceding loop would have run indefinitely until it reached LPC's internal evaluation limit, at which point an error specifying that the evaluation was `too long` would have resulted.

Escapes, *#define*, and *#undef*

When dealing with strings in LPC, you must use certain escape codes to indicate specific types of formatting. There are two escapes of relevant importance: `\n` and `\t`. You saw the `\n` escape in the tidbit of code (called TESTCODE) discussed in the section on operators. `\n` indicates that a carriage return should be inserted where the `\n` appears in a string. The `\t` escape, which seldom is used and is new to you, indicates that a tab (five-space indention) should be inserted where the `\t` appears in a string.

You also need to be familiar with `#define` and `#undef`. The `#define` is used to globally define something. The `#undef` is `#define`'s counterpart and is used to undefine it. Obviously, you can't `#undef` something you haven't `#defined`. It is recommended that `#defines` always consist of capital letters, as this makes them easier to see within the code. Look at the following example:

```
#define PHAEDRUS "Idealistic dreamer"
```

Escapes, `#define`, and `#undef` are all preprocessed. When the code in a file is about to be compiled, it first is processed. During this preprocessing, the values of `\n` (a carriage return), `\t` (a tab), and any `#defines` are substituted for their markers. So, assuming that the preceding `#define` is in a file, the preprocessor will mnemonically replace every occurrence of `PHAEDRUS` with `Idealistic dreamer` until it finds a `#undef PHAEDRUS` statement or reaches the end of the file.

Inclusion and Inheritance

Inclusion and *inheritance* are two ways by which you can incorporate all the aspects of one file into another file. Inclusion of a file is accomplished by using a `#include` statement followed by a filename, as in the following example:

```
#include <living.h>
#include "/obj/living.h"
```

When #include is handled by the preprocessor, it searches through the system's standard include directories (as defined by the gods who tinkered on the driver) if only a filename is given. If it fails to find the file within the standard include directories it searched, an error will result. If, however, you specify a full path to #include (as in the preceding second example), there should be no problem with the inclusion (provided the file exists), as the preprocessor searches the path indicated. This assumes, in both cases, that the file you are including works properly.

The #include statement primarily is used for including files that contain commonly used #defines or other common data. It is considered good form to have any #includes at the beginning of your file, along with your #defines and global variable declarations. Do not use #include for incorporating non-data-type code (that is, code that defines an object) into an object you are building. That's what inheritance is for.

Inheritance is accomplished with the inherit statement followed by a filename. Inheritance *must* be done before any local variables or functions and should be done near the beginning of your file to make it easier to spot, as in the following example:

```
inherit "/obj/monster.c";
```

You should use inheritance whenever you want to incorporate code from one type of object into the file that defines the object you are building. Obviously, if you inherit /obj/monster.c;, as in the preceding example, you should be working on a monster because inheriting monster.c will make all the aspects of monster.c part of the definition of the object on which you are working. Making inheritance of /obj/monster.c, simply put, enables you to use all the global variables and functions from /object/monster.c in the object on which you are working, as if they were a part of the code that you already had written.

Functions

The last and largest component of LPC code is LPC's functions. *Functions* are divided into two groups, known as efuns and lfuns. efun stands for external function. An efun, simply put, is a function that is external to the piece of code you are working on. In other words, it is a function that is defined in the driver or in simul_efun.c and does not, as a result, need to be defined in your code. An lfun, or local function, is a function that is local to a piece of code and must be defined within that piece of code for it to work.

All efuns and lfuns return values. If no value is specified to be returned by a return statement (such as return 1;), then a return of 0 is implied. Table 13.4 shows several types of values a function may return.

Table 13.4. Values a function can return.

Type	*Value Returned*
void	Function returns 0 or 1
int	Function returns an integer value

Type	*Value Returned*
`string`	Function returns a string value
`object`	Function returns a pointer to an object
`varargs`	Function returns any value

Not all functions expect arguments. Those functions that do expect arguments will fail to perform properly if the arguments provided are not of the appropriate type. The types of arguments that can be provided are `int`, `string`, `object`, `mixed`. Arrays of these types also may be used, where mandated.

What an external function returns, its name, and the proper argument types for it are commonly summarized in the following format:

```
void add_action(string func, string cmmd)
```

In the preceding example, `void` indicates that the function returns either `0` or `1` for the function `add_action()`, which expects two arguments, each of which are strings as denoted by `string`. The words `func` and `cmmd` in the preceding example are generally further explained in documentation somewhere on an LPMUD (usually by using `man`). This format is used to describe most of the functions usable in LPC, as it will save time and will familiarize you with both the look of typical manpages as well as type casting.

Functions also may return an array of any of the aforementioned types. Following is an example summary of a function that returns an array:

```
string *get_dir(string dirname)
```

This example follows the same guidelines as the previous example. It differs only in the presence of an `*`, which indicates that the function returns an array of the type that precedes it; in this case, a string.

Some special types modifiers exist, as well. These special types have nothing to do with the value returned by a function, but rather, influence how a function reacts in specific situations. Table 13.5 lists the special type modifiers.

Table 13.5. Special type modifiers.

Type	*Effect*
`static`	Function cannot be called via a `call_other()`
`private`	Function cannot be accessed by any other object inheriting its definition, nor can it be called via a `call_other()`
`nomask`	Function cannot be redefined by a shadow
`public`	Function can be accessed by any other object inheriting its definition, can be called via `call_other()`, and can be redefined by a shadow. All functions default to public unless otherwise specified.

As with variable type modifiers, the preceding modifiers can be mixed and matched when declaring a function. Obviously, you would not declare a function to be both `private` and `public` at the same time. Commonly, `static` and `nomask` are used together to keep any other object from manipulating a function and to prevent redefinition of the function by a shadow. Look at the following example:

```
static nomask int drop(silently) { return 1; }
```

This example ensures that no outside interference can influence the evaluation of `drop(silently)`.

add_action()

```
void add_action(string func, string cmmd)
```

`add_action` is an efun that sets up a local function `func` to be called when a user inputs a command `cmmd`. Look at the following example:

```
add_action( "drop_thing", "drop" );
```

In the preceding example, when `drop wand` is entered, `drop` triggers a call to the local function `drop_thing()`. If `drop_thing()` is found, the argument `wand` (second word of what the user typed) is passed to it for its own use.

Optionally, the second argument `cmmd` can be omitted. If omitted, there must be an associated `add_verb()` or `add_xverb()` statement to accompany the `add_action` statement. Use of `add_verb()` is archaic as the second optional argument of `add_action()` is the verb. It remains supported, but is obsolete. `add_xverb()`, however, is still essential.

You should only call `add_action()` from the `init()` routine in your code.

add_verb()

```
void add_verb(string verb)
```

This efun is connected to the `add_action()` efun. It will set up the command `verb` to trigger a call to the function set up by the previous call to `add_action()`, as in the following example:

```
add_action( "drop_thing" );  add_verb( "drop" );
```

The `add_verb()` efun is, as was previously mentioned, obsolete.

add_xverb()

```
void add_xverb(string xverb)
```

This efun is also connected to the `add_action()` function. It will set up the command `xverb` to trigger a call to the function set up by the previous call to `add_action()`. `add_xverb()` is different from `add_verb()` in that a space is not required between the command and its first argument. Look at the following example:

```
add_action( "emote" ); add_xverb( ":" );
```

Assuming that a function `emote()` is defined to permit you to emote, the preceding example enables you to enter `:smiles.` to `emote` a smile.

all_inventory()

```
object *all_inventory(object ob)
```

The `all_inventory()` efun returns an array of the objects contained in the inventory of the object `ob`, as in the following example:

```
object *allob;
allob = all_inventory( this_object() );
```

The preceding example sets the object variable `allob` equal to an array consisting of pointers to all the objects contained in the inventory of `this_object()`.

allocate()

```
mixed *allocate(int size)
```

This efun allocates an array of `size` elements. The number of elements must be greater than or equal to `0` and must not exceed the standard system maximum (which usually is `1000`).

`allocate()` is hardly needed anymore, as arrays can be added using the `+` operator. Also contributing to the obsolescence of this function is the fact that arrays can be constructed and initialized using the `({ })` format, as in the following example:

```
string *queue;
queue = allocate( 100 );
```

atoi()

```
int atoi(string str)
```

The `atoi()` efun returns an integer if the string `str` is an integer. `0` will be returned if the string `str` is not an integer, as in the following examples:

```
atoi( "42" )
```

```
atoi( "bs" )
```

The first example returns an integer value of `42`, while the second example returns `0`.

`atoi()` is used to convert strings to integers, where possible.

break_string()

```
string break_string(mixed str, int length, mixed indent)
```

This efun breaks a continuous string that contains no newlines (`\n`'s) into a string with newlines inserted at every `length` character.

The `indent` argument is optional. If `indent` is defined and given as an integer, `indent` number of spaces are inserted after every new line. If `indent` is defined and is a string, `indent` is inserted before every new line.

If the first argument is given as an integer instead of a string, `break_string()` will return `0`.

call_other()

```
unknown call_other(object ob, string func, mixed arg1, mixed arg2, ...)
```

This efun calls a function `func` in an object `ob` with the arguments `arg1`, `arg2`, and as many other arguments as you want to pass. The value returned by `call_other()` is that which it received from the function `func` it attempted to call in object `ob`. If `ob` has not yet been loaded, it will be loaded.

`call_other()` can be written syntactically in code to save space. `ob->func(arg1, arg2)` and `"full path of file"->func(arg1, arg2)` are both equivalent to `call_other(ob, "func", arg1, arg2)`. If you want, you can use a string variable in place of `full path of file` when using that format.

If `ob` does not define a function `func`, `call_other()` will return `0`. Look at the following examples:

```
call_other( "/obj/player.c", "drop_thing", "all" );
```

```
"/obj/player.c"->drop_thing( "all" );
```

Both the preceding examples achieve the same thing, which is to call a function, `drop_thing()` in the object whose definition file is `/obj/player.c` and pass the argument `all` to that function. Following is a cleaner way to do the same thing:

```
object ob;
ob = find_object( "/obj/player.c" );
ob->drop_thing( "all" );
```

call_out()

```
void call_out(string func, int delay, mixed arg)
```

The `call_out()` efun calls the function `func` in `this_object()`. The call will take place in `delay` seconds. If the optional argument, `arg`, is supplied, it is passed to `func`. You should note that the efun `this_player()` will not work in a function called via `call_out()`—`this_player()` must either be passed to the function `func` as the argument `arg` or stored in a global variable if it needs to be maintained, as in the following example:

```
call_out( "hit_player", 2, this_player() );
```

call_out_info()

```
mixed *call_out_info()
```

This efun gets information about call_outs that are pending. It returns an array, whose elements are themselves arrays, each of which consists of four elements, as described in Table 13.6.

Table 13.6. Elements of arrays within `call_out_info()` return value.

Element	*Content of Element*
0	Pointer to the object with pending `call_out()`
1	Function being called out
2	Delay remaining until function is called
3	Optional argument passed in `call_out()`

caller()

```
object caller()
```

This efun returns a pointer to the object that called the current function. If `caller()` is called from the parameter list of a function call, it will return the pointer to the object that contains the function being called, *not* a pointer to the object that called the current function.

can_put_and_get()

```
int can_put_and_get(string str)
```

The `can_put_and_get()` lfun returns `true` (1) if it is possible to put something into the object; otherwise, it returns `false` (0). If no return statement is supplied, as with all functions, the default return value will be `0`. If you do not put this function into an object, then it is the same as if the function returned `0`. This means that all objects default to disallowance of the placement of objects inside of them, unless the following statement exists within the file that defines them (either by inheritance, or by having been coded into the object), as shown in the following:

```
can_put_and_get(str) { return 1; }
```

When a player enters `look at` *`XXX`*, the string value of *`XXX`* is passed to `can_put_and_get()` as `str` to test if the player is permitted to look at the inventory of the object. In all other cases, `str` is `0`.

capitalize()

```
string capitalize(string str)
```

The `capitalize()` efun converts the first character in `str` to an upper-case character and then returns the new string. Passing non-string values to this function causes it to fail. It is recommended that you perform a check to ensure that something is a string by using `stringp()` before passing it on the `capitalize()`. Look at the following example:

```
string str;
str = "MUDsex is dumb.\n";
if ( stringp( str ) ) str = capitalize( str );
```

The preceding example declares `str`, sets `str` equal to `MUDsex is dumb.\n`, checks to see that `str` is a string and, if so, capitalizes it. The value of `str` is then set to what `capitalize()` returns, that being `MUDsex is dumb.\n`.

cat()

```
int cat(string filename, int start, int length)
```

The `cat()` efun lists the file found at `filename`. `filename` should be expressed as a full path and may not contain spaces. The optional integer arguments `start` and `length` are used to `cat` different blocks of a file. The `start` parameter indicates the first line number to display. The `length` parameter indicates how many lines to display after the `start` line. `cat()` returns an integer value that is equal to the number of lines that it displayed. If the value returned is less than `length`, then `cat()` reached the end of the file. `cat()` will return a negative value if one or more of the parameters were improperly given.

The `start` parameter also may be a negative number. In this case `length` still indicates how many lines to display from the `start` line, but the file will be `cat`ted from last line to first. In other words, specifying a negative start parameter tells `cat()` to act like `tail()` (reading from the end of the file to the beginning).

catch()

```
mixed catch(string expr)
```

The `catch()` efun is used to trap errors. The expression `expr` will be evaluated and `0` will be returned if there is no error. If there is a standard error, a string containing a leading `*` is returned.

The `throw()` efun can be used to immediately return any error caught by `catch()`.

`catch()` is a CPU-expensive function and should be used with discretion. Generally, `catch()` is used only in places where an error would cause serious problems to the game as a whole.

The following example illustrates the use of `catch()` to trap an error, should one occur, when the function `drop_thing()` is called (with an argument of "all") in `/obj/player.c`.

```
catch( "/obj/player.c"->drop_thing( "all" ) );
```

catch_tell()

```
void catch_tell(string str)
```

The `catch_tell()` lfun is used to enable objects to act when they receive a `tell_object()`, `tell_room()`, or `say()` that contains the string `str`. The object that contains `catch_tell()` must be a living object and must have `enable_commands()` set in it.

clear_bit()

```
string clear_bit(string str, int num)
```

This `efun` returns the new string that results when bit `num` is cleared in the string `str`.

clone_object()

```
object clone_object(string filename)
```

The `clone_object()` `efun` loads a new object from the definition file `filename` and gives the object a unique object number. A pointer to the object is returned. The following example illustrates the proper syntax for `clone_object()`.

```
clone_object( "/obj/torch.c" );
```

command()

```
int command(string str, object ob)
```

`command()` is an efun that, when called, executes `str` as a command in the current player as if it were issued by that player. If the second optional argument `ob` is given, `command()` will execute `str` as a player command in the object specified by `ob`. Generally, functions that are static cannot be called via `command()`. This provides some protection from potential `command()` abuses.

An object must be living and have `enable_commands()` set in order for `command()` to work on it. `command()` returns `1` if it is successful in forcing the execution of `str`, `0` if it fails. The following example shows the proper syntax for use of `command()`.

```
command( "smile", this_player() );
```

cp()

```
void cp(string source, string destination)
```

The `cp()` efun copies the file `source` to the file `destination`. Both arguments should be expressed as a full path and neither may contain spaces.

create()

```
void create()
```

`create()` is an lfun found only on LPMUDs running in native mode. This efun is called only once, when the object being loaded is first created. All major initialization for the object being generated should be done inside of this function. You should note that when `create()` is being processed, the object containing the `create()` statement does not yet have an environment.

LPMUDs of version 2.4.5 or older use `reset()`, not `create()`. This also is true of LPMUD versions 3.0 (or more recent) that are running in compat mode.

create_wizard()

```
string create_Wizard(string name)
```

The `create_wizard()` efun creates a home directory and a castle for a wizard. The directory is created for the wizard name `name`. If archaic-style castles are used, a copy of the definition of a castle will be placed within the newly created directory. Automatic loading of the castle also will be set up if a copy of the basic castle definition is made. The string returned will be the name of the new castle. If an error occurs, `0` is returned.

creator()

```
string creator(object ob)
```

The `creator()` efun returns a string consisting of the name of the wizard that created object `ob`. If the object was not created by a wizard `creator()`, it returns `0`.

crypt()

```
string crypt(string str, string seed)
```

The `crypt()` efun encrypts the string `str` using two characters from the string `seed`. If the `seed` is `0`, then a random `seed` is used.

ctime()

```
string ctime(int clock)
```

The efun `ctime()` evaluates the argument `clock` as the number of seconds since January 1st, 1970, at 0.00 hours and converts it to a human-readable string in the following form:

```
Tue Nov 27 02:022:51 1980
```

If `clock` is not specified, `time()` is used as a default.

deep_inventory()

```
object *deep_inventory(object ob)
```

The `deep_inventory()` efun returns an array of all objects contained by the given object `ob`, including the objects recursively contained in other objects. If the argument `ob` is not given, `deep_inventory()` defaults to using `this_object()` as an argument.

destruct()

```
void destruct(object ob)
```

`destruct()` is an efun that, when called, destroys and removes the definition of object `ob`. The argument also can be a string that is the full path of the definition file from which the object you want to destroy was loaded. After `destruct()` has been called on `ob`, all global variables will be set to an uninitialized state (that is, they will be set to `0`). Following is an example:

```
destruct( this_object() );
```

disable_commands()

```
void disable_commands()
```

`disable_commands()` is an efun that disables the capability of the object in which it was called to use commands normally added by external objects. Normally, external commands are added when external objects enter the immediate environment of the object in which `disable_commands()` was called; `disable_commands()` simply prohibits the addition of commands when addition is supposed to take place. Calling `disable_commands()` will cause a living object to cease to be classified as living.

drop()

```
int drop(int silently)
```

`drop()` is an lfun that is defined by all objects that need to control whether they can be dropped. If `silently` is `true` (that is, `silently` is equal to any value other than `0`), then no message will be written to indicate that the item has been dropped.

All objects default to being droppable (because a function returns `0` unless told to do otherwise by use of the return statement). If you want an object to disallow the dropping of itself, `drop()` should return `1`.

If you make an object that self-destructs when `drop()` is called, you should be sure that `drop()` returns `1`, as an item that was just destroyed cannot also be dropped! Following is an example of the `drop()` command.

```
drop(silently)
{
if ( present( "/obj/curse.c", this_player() ) ) return 1;
}
```

An object whose definition file contains the preceding `drop()` function will not allow itself to be dropped if `/obj/curse.c` (presumably a curse of some sort) is present in the environment of `this_player()` (the person who issued the command that called `drop()`).

enable_commands()

```
void enable_commands()
```

`enable_commands()` is an efun that enables the capability of the object in which it was called to use commands normally added by external objects when they enter the immediate environment of the object in which `enable_commands()` was called. The object in which `enable_commands()` was called also will be marked as living (that is, a call of `living()` on the object will return `true`). `enable_commands()` must be called within the object if your intention is for the object to interact with other players, as this is what makes an object fall into the classification of living.

Avoid calling `enable_commands()` from anywhere in your code, except within the body of `reset()` (or `create()` if the LPMUD on which you are coding is running in native mode). This is because the issuer of the command will immediately be set to the new object.

environment()

```
object environment(object ob)
```

The `environment()` efun returns a pointer to the object that surrounds object `ob`. If no argument is given, then a pointer to the object which surrounds the current object (`this_object()`) is returned. Following is an example:

```
object plr;
   plr = environment( find_player("bodie") );
```

The preceding example declares object `plr` and sets it to a pointer to the object surrounding that object to which `find_player()` (which is looking for a player named "bodie") returns a pointer, if there is one.

exec()

```
int exec(object new, object old)
```

The `exec()` efun is used to shift an interactive user from one object, `old`, to another object, `new`. A function local to `master.c`, `valid_exec()`, is called with the object issuing the `exec()` given as an argument. If `master.c`'s `valid_exec()` accepts the calling object, then the interactive user is switched from the object `old` to the object `new`.

exit()

```
void exit(object ob)
```

The `exit()` lfun is intended for use in rooms and is used only on LPMUDs running in compat mode. `exit()` is called every time a living object `ob` leaves the object (which should be a room) in which `exit()` is defined. After `exit()` has been called, the function `this_player()` will return a random value. You should store `this_player()` in a global variable before `exit()` is called in your room if you need to keep track of it.

explode()

```
string *explode(string str, string char)
```

The `explode()` efun returns an array of strings consisting of each substring of `str` separated by the string `char`. `explode()` is used to break a string into pieces using `char` as the character around which the break is done. The string component `char` is omitted from the pieces. Following is an example:

```
explode( "players/whiplash/workroom", "/" );
```

The preceding example would return an array whose elements are strings in the form of:
`({"players", "whiplash", "workroom"})`

extract()

```
string extract(string str, int from, int to)
```

The `extract()` efun extracts a substring from a string `str` starting at character `from` and terminating the extraction at character `to`. The first character of a string is element `0` of that string, *not* `1`. The last parameter, `to`, is optional. If the last parameter is not specified, then extraction will proceed from character `from` to the end of the string. Following is an example:

```
extract( "MUDding is addictive", 11, 19 );
extract( "MUDding is addictive", 11 );
```

Both of the preceding examples mean the same thing. Extraction as specified preceding will return the string `addictive`.

file_name()

```
string file_name(object ob)
```

The `file_name()` efun gets the file name of the object `ob`. If the object for which you are trying to get a filename is a cloned object, it will not have any corresponding filename. Instead, it will have a new name, which is based on the original file name. Following is an example:

```
file_name( this_object() );
```

This example returns the filename of the object in which `file_name()` is called.

file_size()

```
int file_size(string filename)
```

This efun returns the size of a file `filename`. `filename` should be expressed as a full path and may not contain spaces. If `filename` does not exist, `file_size()` returns `-1`. A value of `-2` is returned if `filename` is a directory. Following is an example:

```
if ( file_size( "/lpMUD.log" ) == -1 )
write("File does not exist.\n");"
```

The preceding example will write `File does not exist.` if the file `/lpMUD.log` does not exist.

file_time()

```
int file_time(string filename)
```

The `file_time()` efun returns the time in seconds since January 1st, 1970, at 0.00, when the file `filename` was last modified. `filename` may not contain spaces and should be expressed as a full path.

filter_objects()

```
mixed *filter_objects(mixed *arr, string func, object ob, mixed extra)
```

The `filter_objects()` efun returns an array holding the items of `arr` filtered through a `call_other ()` to the function `func` in the object `ob` (`ob->fun()`). The function `func` in object `ob` is called for each element in `arr` with that element as a parameter. If a second parameter `extra` is specified, it is sent in each call, as well. If the `call_other()` returns `true`, then the element is included in the array returned by `filter_objects()`. A value of `0` is returned by `filter_objects()` if `arr` is not an array.

find_call_out()

```
int find_call_out(string func)
```

The `find_call_out()` efun finds the next pending `call_out()` for the function `func` in `this_object()`. The amount of delay time left in the `call_out()` is returned by `find_call_out()`. If the function `func` has not been set for a `call_out()`, then `find_call_out()` will return a value of `-1`. Following is an example:

```
if (find_call_out("waste_time")) remove_call_out("waste_time");
```

The preceding example searches `this_object()` for a `call_out()` of the local function `waste_time()`. If a `call_out()` is found, then it will be removed, as per the `then` portion in the preceding example.

find_living()

```
object find_living(string str)
```

`find_living()` is an efun that tries to find the first `living` object that answers to the identity `str` when `id()` is called. If such an object is found, a pointer to the object is returned. If not found, `find_living()` returns `0`. Following is an example:

```
object ob;
ob = find_living( "dwarf" );
```

In this example, if a living object that `id`'s to `dwarf` is found within the game, then `ob` is set to a pointer to the object returned by `find_living()`, that object being the one which `id`'d to `dwarf`.

find_object ()

```
object find_object(string str)
```

`find_object()` is an efun that tries to find an object that has the filename of `str`. This will only work if the object has been loaded. If no object exists that has the filename `str`, then `find_object()`returns `0`. Following is an example:

```
object ob;
ob = find_object( "/obj/master.c" );
```

This example will, if `/obj/master.c` is loaded (and it *must* be for the game to be up!), set `ob` equal to a pointer to it.

find_player()

```
object find_player(string str)
```

The `find_player()` efun attempts to find the player whose name is `str`. If a player object whose name is `str` is found, then `find_player()` returns a pointer to that object. If no player object whose name is `str` can be located, then `find_player()` will return `0`. Following is an example:

```
object ob;
ob = find_player( "stingray" );
```

first_inventory()

```
object first_inventory(object ob)
```

The `first_inventory()` efun returns a pointer to the first object in the inventory of object `ob`. Following is an example:

```
object ob;
ob = first_inventory( this_object() );
```

function_exists()

```
string function_exists(string func, object ob)
```

The `function_exists()` efun returns the filename of the object that defines the function `func` in object `ob`. The returned value can be something different than what is returned by `file_name(ob)` as a result of object `ob` inheriting some other file. In this case, the `filename` of the inherited file will be returned. `function_exists()` will return `0` if the function `func` is undefined.

get()

```
int get()
```

`get()` is an lfun that is defined by all objects that need to control whether they can be picked up. When a player enters `get` *xxx*, `id()` is called with the argument of *xxx*. Assuming that an object is found that responds to the `id` of *xxx*, *and* assuming that the object is within the environment of the player, `get()` is called. `get()` should return `1` if it is possible to pick up the object.

All objects default to refusing to be picked up (because a function returns `0` unless told to do otherwise by use of the return statement). This is so that certain objects, such as `players`, cannot be inadvertently picked up. Following is an example:

```
get()
{
if ( this_player()->query_level() >= 21 ) return 1;
}
```

An object whose definition file contains the preceding `get()` function will allow itself to be picked up if the value returned by a `call_other()` to the function `query_level()` in `this_player()` (the person who issued the command that called `get()`) is greater than or equal to `21`. Assuming level `21` to be the minimum level of a wizard, this could be used in an object that only a wizard may pick up.

get_dir()

```
string *get_dir(string dirname)
```

The `get_dir()` efun returns a string array containing the names of all files located within the specified directory `dirname`. The string `dirname` should be a full path with a trailing slash followed by a period. Following is an example:

```
get_dir( "/obj/." );
```

This example returns a string array whose elements are all the filenames in the directory `/obj`.

get_localcmd()

```
string *get_localcmd(object ob)
```

`get_localcmd()` is an efun that returns a string array of all local commands added to the object `ob`. If no argument is given, then `get_localcmd()` uses `this_object()` by default.

grab_file()

```
string *grab_file(string filename)
```

The `grab_file()` efun loads the entire file `filename` into a string array. Each element of the returned array is a separate line of the file `filename`. There is an internal maximum limit to the size of the array, which typically is 1000 elements.

heart_beat()

```
void heart_beat()
```

`heart_beat()` is an lfun that is automatically called by the game every two seconds. When using `heart_beat()`, you should make sure that it is running only when necessary, as `heart_beat()` consumes many game resources. The `heart_beat()` of an object may be turned off and on by using the `set_heart_beat()` efun. If there is an error somewhere in the `heart_beat()` routine, `heart_beat()` automatically will be turned off and will not be able to be restarted with `set_heart_beat()`. The object must be fixed and recompiled.

If the definition file of an object contains `heart_beat()` and the object also is defined to be living, the function `this_player()` will return `this_object()` (that is, your non-living code is now considered a player because it is living and has a heart beat!). If the object is non-living (because `enable_commands()` wasn't called in the object's `create()` or `reset()`), `this_player()` will return `0`. Following is an example:

```
object owner;
create()
{
  owner = this_player();
  set_heart_beat(1);
}
heart_beat()
{
  tell_object( owner, "You hear your own heartbeat\n" );
}
```

In the preceding example, the `heart beat` is set when `create()` is called (when the object is first loaded) and `owner` is set to `this_player()`, whose value is `0` because you did not call `enable_commands()` in `create()`. When `heart_beat()` is called, `owner` (in this case no one) is told `You hear your own heartbeat`. Had `enable_commands()` been called in the `create()` (or `reset()` for LPMUDs of version 2.4.5 or LPMUDs running in compat mode), the object would have told itself that it hears its own heartbeat!

id()

```
int id(string str)
```

The `id()` lfun enables an object to identify itself. If the string `str` matches a given `id` for the object, then a non-zero value is returned; otherwise, `id()` returns `0`. Following is an example:

```
id(str)
{
  return str == "sword" || str == "Excaliber";
}
```

In the preceding example, if the value of `str` is equal to `sword`, *or* if the value of `str` is equal to `Excaliber`, the result of the check of equivalency is returned. (The result would be `0` for either check if non-equivalency were found, `1` for equivalency. Only one of these checks needs to be true for a non-zero value to be returned by `id()`, signifying that the string `str` IS a valid `id` for the object.)

implode()

```
string implode(mixed *arr, string del)
```

`implode()` concatenates all strings found in array `arr`, with the string `del` between each element. Only strings from the array are used.

explode()

```
string implode(mixed *arr, string add)
```

The `explode()` efun concatenates all strings found in array `arr`, placing the string `add` between them. The value of `add` may be `""`, which is an empty string. If the array `arr` contains integers as well as strings, only the strings will be used for concatenation. Following is an example:

```
string str;
str = implode( ({"Macron", "is", "Fodder"}), " " );
```

The preceding example would set `str` to `Macron is Fodder`, the string value returned by `implode()`. Following is an example:

```
string str;
str = implode( ({"Omni", "potence"}), "" );
```

This example illustrates the use of an empty string for implosion. In this case, `str` is set to `Omnipotence`.

index()

```
int index(mixed arr, mixed target, int offset, int increment)
```

`index()` is an efun. The first parameter `arr` also can be an array. The position of `target` inside `arr` is returned by `index()`. If `target` is not in `arr`, `index()` returns `-1`. When `offset` is specified, `index()` will search from `offset` to the end of `arr`, be it a string or an array. When `increment` is specified, the search is done in steps of `increment`.

If `arr` is a string, `index()` searches for the first instance of the character `target` as if the string were an array of characters. (Conceptually, that's precisely what a string is!)

inherit_list()

```
string *inherit_list(object ob)
```

The `inherit_list()` efun returns a string array of all files that are inherited in the specified object `ob`. If no argument `ob` is given, then `inherit_list()` will use `this_object()` by default.

init()

```
void init()
```

`init()` is an lfun whose main purpose is to facilitate the set up of `add_action()`, singularly or en mass. `init()` is called when two objects move near one another, provided that one of the objects is marked as living.

Basically, if object `A` is living and moves into object `B`, `init()` is called in object `B` and `this_player()` is set to return a pointer to object `A`. For every living object `C` inside of object `B`, `init()` is then called in object `A` with `this_player()` set to return a pointer to object `C`. Then, again for every living object `C` inside of object `B`, `init()` is called in object `C` with `this_player()` set to return a pointer to object `A`. Finally, `init()` is called in object `A` with `this_player()` set to return a pointer to object `B`.

Simply put, when a living object enters another object (in which are located several more objects), each object present calls `init()` in the object it perceives to be the newcomer to the set of objects now present. Following is an example:

```
init()
{
  add_action( "drop_thing", "drop" );
  add_action( "get_thing", "get" );
}
```

The preceding example is how a sample of the `init()` in `player.c` might look. It sets up two commands, `get` and `drop`, with the appropriate functions to search when either is invoked.

input_to()

```
void input_to(string func, int flag)
```

`input_to()` is an efun that enables the next line of user input to be sent to the local function `func` as an argument. The input line will be passed directly to the function `func` without being modified in any way. The function `func` will not be called immediately. Execution of `func` will not begin until the current execution from which `input_to()` was called has terminated and the player has given a new command. If `input_to()` is called more than once in the same execution, only the first call has any effect.

If the optional argument `flag` has a non-zero value, then the line entered by the player will not be echoed, preventing both the originator of the line as well as any would-be snoopers from seeing it. Following is an example:

```
input_to( "more_file" );
```

The preceding example will, upon terminating the function in which `input_to()` was called, send the next line of entered text to the function `more_file`.

interactive()

```
int interactive(object ob)
```

The `interactive()` efun returns `1` (`true`) if the object `ob` is connected to a socket. `interactive()` will return `0` if the object `ob` is not connected to a socket. Following is an example:

```
object ob;
ob = find_player( "stringray" );
if ( ob && !interactive( ob ) ) destruct( ob );
```

The preceding example declares `ob`, sets `ob` to a pointer returned by calling `find_player()` on the player whose name is `stringray` and, if a pointer to the object that is the player named `stingray` is found and that object is not interactive, destructs the object pointed to by `ob`.

intp()

```
int intp(mixed arg)
```

`intp()` is an efun that returns `1` if `arg` is an integer, and `0` if it isn't.

living()

```
int living(object ob)
```

The `living()` efun returns `1` if `ob` is a living object, and `0` if it isn't.

log_file()

```
void log_file(string filename, string msg)
```

`log_file()` is an efun that writes a message `msg` to a log file `filename`. The file will be opened, written to, and then closed (if the file already exists, it will be appended to; if not, the file will be created and then appended to.) All log files are located in `/log`.

long()

```
void long(string str)
```

`long()` is an lfun that enables an object to write a description of itself.

The string argument `str` is optional. If `str` is specified, then the description of that argument is printed. For `str` to be accepted, it must return `true` when passed to `id()`. Following is an example:

```
long()
{
  short();
  write( "An obsidan orb.  It seems to absorb all light.\n" );
  return 1;
}
```

In the preceding example, assuming that a function `short()` is defined and that `orb` is a valid `id` for the object, when `long()` is called it will call `short()` and then write `An obsidian orb. It seems to absorb all light.` (with a carriage return as specified by `\n`).

lower_case()

```
string lower_case(string str)
```

The `lower_case()` efun converts all characters in `str` to lowercase and then returns the new string. Passing non-string values to this function causes it to fail. It is recommended that you perform a check to ensure that something is a string by using `stringp()` before passing it to `lower_case()`. Following is an example:

```
string str;
str = "ABSOLUT DRINKS TOO MUCH.\n";
if ( stringp( str ) ) str = lower_case( str );
```

The preceding example declares `str`, sets `str` equal to `ABSOLUT DRINKS TOO MUCH.\n`, checks to see that `str` is a string and, if so, converts it to lowercase characters. The value of `str` then is set to what `lower_case()` returns, that being `absolut drinks too much.\n`.

ls()

```
void ls(string filename)
```

The `ls()` efun lists files in `filename`. The string `filename` may be expressed as a full path and cannot contain periods or spaces.

map_array()

```
mixed *map_array(mixed *arr, string func, object ob, mixed extra)
```

`map_array()` is an efun that returns an array holding the items in `arr` mapped through a `call_other()` to the function `func` in the object `ob`. The function `func` in object `ob` is called for each element in `arr` with that element as a parameter. An optional second parameter `extra` is sent in each call if specified.

The value returned by the `call_other()` `(ob->func( arr[index], extra ))` replaces the existing element in `arr`. If `arr` is not an array, then `map_array()` returns `0`.

member_array()

```
int member_array(mixed item, mixed *arr)
```

The `member_array()` efun returns the index of the first occurrence of `item` in `arr` if an instance is found. If not, then `-1` is returned. Following is an example:

```
member_array( "morpheus", ({ "morpheus", "sly", "elric" }) );
```

The preceding example would return a value of `0`, as `morpheus` is the first element (which always has an index of `0`) in the indicated array.

mkdir()

```
void mkdir(string dirname)
```

The `mkdir()` efun creates the directory `dirname`. The string `dirname` should be expressed as a full path and may not contain spaces.

move_object()

```
void move_object(object item, object dest)
```

`Move_object()` is an efun that moves the object `item` into a destination object `dest`. Following is an example:

```
move_object( clone_object( "/obj/torch.c" ), this_object() );
```

The preceding example `clones` a torch and moves it into the object in which `move_object()` was called.

next_inventory()

```
object next_inventory(object ob)
```

The `next_inventory()` efun returns a pointer to the next object that is located in the same inventory as `ob`. If the object `ob` is moved by `move_object()`, then `next_inventory()` will return an object from the new inventory.

notify_fail()

```
void notify_fail(string str)
```

The `notify_fail()` efun stores `str` as the message to be given in the event of an error when a verb (the command both players and wizards type) is used. This message will be given instead of the standard default error message `What?`. If `notify_fail()` is called more than once, then only the last call is used.

You should call this function inside of a local function you have defined. Commonly, `notify_fail()` is called just before the return statement.

object_time()

```
int object_time(object ob)
```

The `object_time()` efun returns the time in seconds since January 1st, 1970, at 0.00, when the object was created.

objectp()

```
int objectp(mixed arg)
```

The `objectp()` efun returns `1` if `arg` is an object, and `0` if it isn't.

pointerp()

```
int pointerp(mixed arg)
```

`Pointerp()` is an efun that returns `1` if `arg` is an array, and `0` if `arg` is not an array.

present()

```
object present(string str, object ob)
```

The `present()` efun searches the inventory of the current object (`this_object()`) as well as the inventory of the environment of the current object. The search is conducted for an object that `id`'s to the string `str` and a pointer to the object is returned if such an object is found. `Present()` will return `0` if no object that `id`'s to `str` is located.

Optionally, the first argument, `str`, may be given as an object instead of a string. (It is recommended that you use an object for the first argument if possible, as `present()` searches more efficiently when given an object for the first parameter.) Additionally, a second optional argument, `ob`, which is the environment to search rather than the current object's inventory or environment, may be specified. If `ob` is given, `present()` searches only the inventory of `ob`, not its environment. Following is an example:

```
object item;
item = present( "circlet", this_player() );
```

This example searches the inventory of `this_player()` for an object that id's to `circlet`. If an object in which `id()` returns `1` for the string `circlet` is located, then `present()` returns a pointer to it. If not, present returns `0`. The object variable `item` is then set to the value returned by `present()`.

previous_object()

`object previous_object()`

`previous_object()` is an efun that returns an object pointer to the object that did a `call_other()` to the current object, if any. If no object did a `call_other()` to the current object, then `0` is returned. If, however, the `call_other()` originated within the current object itself (`this_object()`), or an object destroyed by `destruct()`, the value returned by `previous_object()` will be `0`. Following is an example:

```
test_if_me()
{
  object prev;
  if( (prev=previous_object()) && prev != this_object() )
    {
    log_file( "ILLEGAL", file_name( prev ) + " attempted "+
     "to call test_if_me!\n" );
    return 0;
    }
  return 1;
}
```

The preceding is a sample function `test_if_me()`. In it, `prev` is declared as an object variable. The first expression in the `if()` statement uses parentheses to ensure that what is within them (the setting of `prev` to the object pointed to by `previous_object()`) is processed first. Once the first expression of the `if()` is evaluated for truth or falsehood (truth being `prev` having been set to a pointer to an object, and falsehood being `prev` having been set to `0`), the second expression of the `if()` statement is evaluated, provided the first expression was true. The second expression checks to see if the object pointed at by `prev` is not the same as `this_object()`. If both expressions of the `if()` statement are `true`, then the clause (which is inside the braces) is evaluated. In it, `log_file()` is used to write the filename of the calling object `prev` and states that it `attempted to call test_if_me!`. Evaluation of the function `test_if_me()` then ends, returning a value of `0`. In the event that one expression in the `if()` statement is `false`, however, the `then` clause is ignored and `test_if_me()` terminates, returning `1`.

NOTE

The preceding example integrates a number of coding steps you have already seen (along with a few you haven't seen yet) to demonstrate the order in which code is executed, as well as the use of `previous_object()`. Don't be alarmed if it doesn't quite make sense to you yet; the more examples you are exposed to, the clearer they will become.

process_string()

```
string process_string(string combinestring)
```

The `process_string()` efun processes a string by replacing specific syntactic patterns with what is returned when the pattern is interpreted as a function call description. Syntactic patterns are in the following form:

```
@@function[:filename][¦arg1¦arg2....¦argN]@@
```

This is interpreted as the following call:

```
filename->function(arg1,arg2,..., argN)
```

`process_string()` will not recurse over returned replacement values. If a function returns another syntactic pattern, that description will *not* be replaced.

All such occurrences in `combinestring` are processed and replaced if the return value is a string. If the return value is not a string, then the pattern will remain unreplaced.

Note that both the filename and arguments are marked optional with the brackets and that the brackets are not included in the actual pattern. Following is an example:

```
foo(string str)
{
  return "FOO"+str+"R";
}
func()
{
  write( process_string( "@@foo¦BA@@")+"\n" );
}
```

In this example, the `func` prints the string `FOOBAR` to the screen called.

`process_string()` is a seldom used (if ever) efun, as its functionality can be duplicated with less confusion (and a lower chance of errors) in other ways.

Following is a potential use of `process_string()`:

```
write( "@@query_name:/obj/monster#123@@ is following you!\n" );
```

Assuming that `/obj/monster#123` exists in the game and has the name `Karelle`, the string written to the screen would be `Karelle is following you!`. If `/obj/monster#123` does not exist, then an error will be displayed.

process_value()

```
mixed process_value(string calldescription)
```

The `process_value()` efun gets the replacement of a syntactic pattern. The pattern is of the following form:

```
function[:filename][¦arg1¦arg2....¦argN]
```

This is interpreted as the following call:

```
filename->function(arg1,arg2,..., argN)
```

Unlike `process_string()`, the value returned by `process_value()` can be of any type.

As with `process_string()`, you should note that both filename and arguments are marked optional with the brackets and that the brackets are not included in the actual pattern.

query_auto_load()

```
string query_auto_load()
```

`query_auto_load()` is an lfun that should be defined in any object that needs to be cloned and moved to the player when he or she logs. Typically, the following rules are observed with auto-loading items:

- Auto-loading items should not have weight.
- Auto-loading items should prevent players from dropping them.
- Auto-loading items should be loaded into memory before the player logs in, or else the player will not be given a copy.
- Auto-loading items should not be usable player objects such as weapons, armor, or healing objects.

`query_auto_load()` must return a string in the format `filename:arg`. The `filename` is the definition that will be cloned while the `arg` is a string that will be sent as an argument to the function `init_arg()`, which is local to the object whose definition file is `filename`. The `arg` can be an empty string. `init_arg()` is where the object to be `auto_loaded` does its own internal configuration.

The concept behind this function is that a player can have a curse or a badge of membership that is always with him or her, even if he or she quits. On many LPMUDs, the need for this has been outdated for some time. Where `query_auto_load()` has not been outdated, you may find that it works differently from what has been previously described, as someone may have tweaked the driver to make `query_auto_load()` more useful. Also, the preceding guidelines mentioned for auto-loading items is a general set of rules that have a wide range of variance from MUD to MUD.

query_host_name()

```
string query_host_name()
```

The `query_host_name()` efun returns a string that is the name of the machine on which the game is running.

query_idle()

`int query_idle(object ob)`

`Query_idle()` is an efun that returns the number of seconds for which a user has been idle.

query_ip_name()

`string query_ip_name(object ob)`

`query_ip_name()` is an efun that returns the `ip`-name of the machine from which object `ob` (presumably a player or other user) is logged in. An asynchronous process called `hname`, which runs in parallel with the game driver, is used to find out ip-names. If any failures occur while `query_ip_name()` is locating the `ip`-name for `ob`, then the `ip`-number is returned instead.

query_ip_number()

`string query_ip_number(object ob)`

`query_ip_number()` is an efun that returns the `ip`-number of the machine from which object `ob` (presumably a player or other user) is logged in.

query_level()

`int query_level()`

Generally, all living objects must define the `query_level()` lfun to return a non-negative, non-zero value. On modern LPMUDs, `query_level()` may be limited to use only with monsters and players, as wizards generally are not considered players, today. Historically, `query_level()` played an important part in LPMUD security operations, as levels were the differentiating factors among players (and wizards were then considered players). In the past, an apprentice wizard was usually level 20, a full wizard with a castle was usually level 21, and higher-level wizards had significantly higher levels to match their inflated egos.

query_load_average()

`string query_load_average()`

The `query_load_average()` is an efun that returns a string indicative of the workload of the game driver in the form of `0.68 cmds/s, 29.40 comp lines/s`. Both the average number of commands per second and the average number of compiled lines per second are computed based on the number of commands and compiled lines (respectively) handled over the last 15 minutes of game time.

query_name()

```
string query_name()
```

`query_name()` is an lfun that returns a string consisting of a designated name for the item being queried. All objects handled or seen by players must have a name so that they may be easily referenced by players. The choice of the name that players will see affects what the object should `id` to, as well as an object's name should be enough for the player to figure out what to type to manipulate the object. Following is an example:

```
query_name() { return "wand of riches"; }
```

An object that has the preceding example in it should probably `id` to `wand` and possibly `wand of riches` because most people who see a `wand of riches` would want to enter `get wand` to procure it. At the very least, a player should be able to enter `get` followed by the name of the item.

query_prevent_shadow()

```
void query_prevent_shadow()
```

The `query_prevent_shadow()` lfun, when defined in an object, simply disables the capability of the object to be shadowed. This function is only used on LPMUD versions 3.0 or more recent that are running in either native or compat mode. Following is an example:

```
query_prevent_shadow() { return 1; }
```

query_snoop()

```
object query_snoop(object snoopee)
```

`query_snoop()` is an efun that returns an object pointer to the object (presumably a user) is snooping another user `snoopee`. Generally, use of this function is restricted in some way, as high level wizards often decide they do not want the extent of their voyeuristic tendencies to be known.

query_verb()

```
string query_verb()
```

The `query_verb()` efun returns a string that is the name of the current command, or `0` if the function in which it is called is not executing from a command. `query_verb()` often is used to enable the `add_action()` of several commands to the same function. Following is an example:

```
init()
{
  add_action( "close_door", "close" );
  add_action( "close_door", "shut" );
}
close_door(str)
```

```
{
  if ( str != "door" ) return;
  write( "You " + query_verb() + " the door.\n" );
  seal_door();
  return 1;
}
```

The preceding example enables the `add_action()` of two commands, `close` and `shut`, to the function `close_door()`, in `init()`. When `close_door()` is called by a player entering either `close` or `shut`, `close_door()` checks to make sure that the argument supplied to the verb is `door`. If the argument supplied is not `door`, `close_door()` terminates evaluation, returning `0`. (A return statement with no value behind it is the same as returning `0`.) Otherwise, evaluation continues within `close_door()`. If the player entered `shut door`, the player would see `You shut the door`. If the player entered `close door`, the player would see `You close the door`. A local function `seal_door()` (not shown here), presumably where the code that actually makes the door look and be closed exists, is then called to make the event occur, after which a value of `1` is returned, terminating evaluation.

random()

`int random(int num)`

The `random()` efun returns an integer value in the range of `0` to `num-1`. Following is an example:

```
int n;
n = random(5)
```

This example returns a number ranging from `0` to `4`.

read_bytes()

`string read_bytes(string filename, int fromchar, int tochar)`

`read_bytes()` is an efun that returns the contents of the file `filename`. If the optional arguments `fromchar` and `tochar` are given, then the function returns the contents from byte `fromchar` to (and including) the byte `tochar`. If `fromchar` is specified but `tochar` is not, `read_bytes()` will read from `fromchar` until it has read the last byte in the file. If `fromchar` is negative, `read_bytes()` will read the file backwards, treating the last byte in the file as if it were the first.

There is a maximum limit to the number of bytes that can be read. This limit is defined when the game driver is compiled and is normally 50,000 bytes.

read_file()

`string read_file(string filename, int startline)`

`read_file()` is an efun that reads a line of text from `filename` and returns it as a string. The argument `startline` is the line number of the file `filename` that you want to read into a string. The filename argument may be expressed as a full path. If you do not specify a startline or startline is a non-positive integer value, then read_file() will return 0. If you try to read past the end of a file (startline is greater than the last line number's value, for example), again, then read_file() will return 0.

remove_call_out()

```
int remove_call_out(string func)
```

The remove_call_out() efun removes the next pending call_out() for function func in the current object (this_object()). remove_call_out() returns the time left in the call_out() before it was removed. If the function func has not been set for a call_out(), then -1 is returned.

rename()

```
void rename(string from, string to)
```

rename() is an efun that renames the file from to the name to. If from is a file, then to may be a file or a directory. If from is a directory, then to *must* be a directory. If to is an already-existing directory, then from will be placed within that directory and maintain its original name. Upon successful completion of rename(), 0 is returned. In the event of an error, 1 is returned.

reset()

```
void reset(int arg)
```

reset() is an lfun that is called internally by the game driver approximately every 45 minutes. For LPMUD versions 2.4.5 (or older) and LPMUD versions 3.0 (or more recent) that are running in compat mode, reset() is called when an object is loaded, with an argument of 0 being passed to it. After the initial reset(), this function will be called by the game driver periodically, but with an argument of 1. LPMUDs running in native mode do not bother passing an argument to reset(), as there is no need to distinguish between the first call to reset() and any successive calls due to the existence of the lfun create().

If a room creates objects within its reset(), it should check to make sure that there aren't already similar objects present before creating more of them. This avoids creating multiple duplicates of the same objects.

restore_object()

```
int restore_object(string filename)
```

The restore_object() efun restores the values of all variables for the current object from the file filename. It is permissible to express filename as a full path. It is not, however, permissible to have a period or spaces in filename. Restore_object() returns 1 if it successfully restores the variables saved in `filename`. Variables that have the type modifier `static` will not have been saved if `save_object()` has been called to save the variables to `filename` and, thus, will not be restored.

If `inheritance` is used in the object in which the `restore_object()` function is called, it might be possible that a variable will exist with the same name in more than one place. When restoring, only the last occurrence of such variables will be restored when multiple entries are encountered. To avoid this problem, a unique name should be used on every non-static global variable within an object. Variables local to functions are not saved by `save_object()` nor are they restored by `restore_object()`, as the variables exist only in the course of function evaluation.

rm()

```
void rm(string filename)
```

The `rm()` efun deletes or removes the argument `filename`. `filename` may be expressed as a full path. It is not legal to have spaces in `filename`.

rmdir()

```
void rmdir(string dirname)
```

`rmdir()` is an efun that removes or deletes the directory `dirname`. `dirname` may be expressed as a full path and may not contain spaces.

save_object()

```
void save_object(string filename)
```

The `save_object()` efun saves the values of all non-static global variables of the current object (`this_object()`) in the file `filename`, which may be expressed as a full path. Variables local to functions will not be saved by `save_object()`, as they exist only during the evaluation of functions. It is not legal to have periods or spaces in `filename`. For successful completion of a `save_object()` call, the object attempting to perform the call must be able to write to the directory in which it is trying to save. `Save_object()` returns `1` upon successful completion of the save. It returns `0` if it encounters an error.

say()

```
void say(string str, object ob)
```

The `say()` efun sends a message `str` to all players in the same object. If the optional argument `ob` is specified, then `str` is sent to all players *except* the object specified by `ob`. If `ob` does not point to a player, then it will be ignored.

say() behaves differently if called from heart_beat(). In this case, the string str is sent to all objects in the same environment of the object that calls say(). Following is an example:

```
init()
{
  add_action( "speak", "say" );
  add_action( "speak" ); add_xverb( "'" );
}
speak(str)
{
  if ( !str ) return;
  say( this_player()->query_name() + " says: " +
    capitalize( str ) + "\n", this_player() );
  return 1;
}
```

The preceding example enables the add_action() of two commands, say and ', to the function speak(). The ' command is an xverb, so no space is needed between the command and its argument. When the appropriate verb or xverb is used, speak() is called. speak() terminates its evaluation, returning 0, if no argument follows the given command. As per the preceding, assuming your name to be Kensho and you entered say LPC makes IRCII look like tinker toys, every player, except you, who is in the same object that you are in, would see Kensho says: LPC makes IRCII look like tinker toys. The speak() function then would terminate execution by returning 1.

set_bit()

```
string set_bit(string str, int num)
```

The set_bit() efun returns a string of the results of setting bit num string str. The original string in which the bit is set is not modified.

set_heart_beat()

```
void set_heart_beat(int flag)
```

The set_heart_beat() efun enables or disables the calling of heart_beat() by the game in the current object (this_object()). If flag is 1, then the game will call heart_beat() every two seconds. If flag is 0, then the game will cease to call heart_beat() until instructed to do so by another set_heart_beat(1) call. You should *always* set_heart_beat(0) when the heart_beat() is not needed, as this helps reduce system overhead.

set_light()

```
int set_light(int lightlvl)
```

An object is, by default, dark. It can be set to a state of not being dark by calling set_light(). The integer argument lightlvl controls how bright the lighted object is. On most LPMUDs, this makes absolutely no difference, as an object is either lit or it isn't. However, LPMUDs with a high degree of sophistication may alter how much of the long() of an object you see based on the intensity of light. Calling `set_light()` with a non-positive integer value darkens the object—the more negative the number, the darker the object is considered to be.

The environment surrounding a lit object also is considered to be lit. The value that `set_light()` normally returns is the sum total amount of light given off by all objects in one location that radiate light.

Note that the value of the argument `lightlvl` is cumulative. If you call `set_light(1)` in an object three times and then call `set_light(0)` in the same object, then the returned value on the `set_light()` you just performed would be `3`. This is true of negative values for `set_light()`, as well.

shadow()

```
object shadow(object ob, int flag)
```

`shadow()` is an efun that works in two specific ways: as a means of applying a shadow to `ob`, and as a means of querying whether `ob` is shadowed.

Shadows themselves do not have an environment. Simply put, it means you cannot go into a shadow, put something else into a shadow, or put a shadow into some other object the way you can with all other objects. (This is why they are called shadows!)

Instead, you must apply a shadow to an object. Once you apply a shadow to an object, there are only two ways to remove it: call `destruct()` on the shadow or call `destruct()` on the object that is being shadowed.

Once you apply a shadow to an object, it is used to redirect calls to functions. If object `A` is applied as a shadow to object `B`, then all `call_other()`s made to object `B` are redirected to object `A` and processed within it. If object `A` does not define the function to which the `call_other()` was made, then the `call_other()` is passed on to object `B` as if everything were normal. In fact, the only object that can `call_other()` into object `B` once it has been shadowed, is the shadow itself (object `A`). Object `B` (the object to which the shadow was applied) cannot even make a `call_other()` into itself without first being routed through object `A`, the shadow.

Before you can apply a shadow to an object, it must be prepared. The shadow must be defined in a file somewhere and be able to be initialized via a `call_other()` because applying a shadow to an object consists of making one object "be the shadow" of another object. The way you do this is to write a local function in a file that will become the shadow. The function needs to expect an object for an argument. Once this is done, write any functions that you want the shadow to affect. In those functions, the code you use should be adequate to the task of changing the behavior of the functions you want to affect. Note, however, that a function of type `nomask` will not be affected if the function is redefined in a shadow, as declaring a function to be of type `nomask` will disallow shadows from affecting it. Following is an example of a shadow definition file:

```
object owner;

start_shadow( ob )
{
owner = ob;
shadow( owner, 1 );
}

long()  { return; }


short() { return; }
```

The preceding example is a working object made by a user's shadow. A global object variable `owner` is declared to keep track of who is to be shadowed. The function `start_shadow()` is used to apply this shadow to a user. Once applied, what is normally returned by the lfuns `long()` and `short()` in the user to which this shadow is applied will be altered. Instead of seeing a short and long description of the user when you look at him or her, you will see nothing. This shadow renders the object that has been shadowed invisible to people who can't locate him or her with wizard tools. To start this shadow, you would use such a wizard tool to make a `call_other()` to the function `start_shadow()` in the preceding object, passing it a pointer to the object that is the user you want to shadow. Note the syntax of the `shadow()` statement. The second argument is a `1`. The `1` indicates that `shadow()` is to apply a shadow to someone. Had the value of the second argument to the `shadow()` function been a `0`, `shadow()` would not have applied the shadow to the specified user, but rather, would have returned a value to indicate whether or not the specified user was shadowed.

That leads you to `shadow()`'s second use. Calling `shadow( ob, 0 )` causes `shadow()` to return a pointer to the object that is shadowing object `ob`. If `ob` has no shadow, then `shadow()` returns `0`. Following is an example:

```
object ob, shad;
ob = find_player( "zamboni" );
if ( ob && (shad = shadow( ob, 0 )) ) destruct( shad );
```

This example illustrates the use of `shadow()` for the purpose of locating a shadow on a user. Here, `ob` is set to a pointer to the object that is the player "zamboni." If `ob` exists and there is a shadow applied to `ob`, then `shad` is set to a pointer to the object that is the shadow, after which `destruct()` is called on `shad`.

On most LPMUDs, an object that defines a local function `query_prevent_shadow()` to return `1` cannot be shadowed at all.

NOTE

`shadow()` only works on LPMUD versions 3.0 or more recent that are running in either native or compat mode.

short()

```
string short()
```

`short()` is an lfun that all objects must have. This function returns a string, which is a short message that describes it. Invisible objects will return `0`. Following is an example:

```
short() { return "A socratic teacher.\n"; }
```

shout()

```
void shout(string str)
```

The `shout()` efun sends a string `str` to all users and returns `1`.

sizeof()

```
int sizeof(mixed *arr)
```

The `sizeof()` efun returns the number of elements in the array `arr`.

slice_array()

```
mixed *slice_array(mixed *arr, int from, int to)
```

`slice_array()` is an efun that returns an array that is a slice of the argument array `arr`, starting at the index `from` and ending at the index `to`. As the index to the first element in an array is `0`, neither `from` nor `to` can be negative values. If `arr` is not an array or either index (`from` or `to`) is out of bounds (that is, it's less than zero or greater than the numerical value of the last index), `0` is returned.

sscanf()

```
int sscanf(string str, string fmt, mixed arg1, mixed arg2, ..., argN)
```

The `sscanf()` efun parses a string `str` using the format `fmt`. `fmt` can contain strings separated by `%d` and `%s`. Every `%d` and `%s` corresponds to its respective variable, `arg1`, `arg2`, to `argN`.

`%d` indicates that the value to be scanned out should be treated as an integer, while `%s` indicates that the value should be treated as a string. `Sscanf()` returns the number of matched `%d` and `%s`. If a matching fails, the variable to which the match would have been set is left unchanged. Following is an example:

```
string str, what, whom;
str = "get all from corpse";
if ( sscanf( str, "get %s from %s", what, whom ) != 2 )
  write( "Get <what> from <whom>?.\n" );
else
  write( "You got " + what + " from " + whom + "!\n" );
```

The preceding example illustrates the use of `sscanf()` to do two things at once: determine whether the string `str` matches the pattern `fmt`, and if so, scan pieces of the format (symbolized by `%s`, here) into variables. The string `str` is set to `get all from corpse`. `sscanf()` then compares `str` to `get %s from %s`. If it finds that there is a match between the solid parts in the format (that is, the words `get` and `from`), then `sscanf()` puts each `%s`, in order from left to right, into the string variables `what` and `whom` that were given as arguments, again, from left to right. `sscanf()` then returns a value. Since, in this case, there *should* be a match, the value returned should be equal to `2`. If it doesn't equal `2`, `Get <what> from <whom>?` is written to the screen. Otherwise, `You got all from corpse!` will be written to the screen.

stringp()

```
int stringp(mixed arg)
```

The `stringp()` efun returns `1` if `arg` is a string, `0` if it isn't.

strlen()

```
int strlen(string str)
```

The `strlen()` efun returns the number of characters in string `str`.

tail()

```
void tail(string filename)
```

`tail()` is an efun that, when called, prints out the end of the file `filename`. `filename` should be specified as a full path and may not contain spaces. There is no specific number of lines that are printed to the screen. Instead, a maximum of 1000 bytes is printed. `tail()` returns `1` after successfully tailing `filename`.

tell_object()

```
void tell_object(object ob, string str)
```

The `tell_object()` efun sends a message `str` to object `ob`. If `ob` is not a user, `str` will be intercepted by the local function `catch_tell()` to be manipulated and acted upon by the object, assuming that `catch_tell()` is defined in the object. `tell_object()` returns `1` if it was able to send `str` to `ob`, or `0` if `ob` could not be located.

tell_room()

```
void tell_room(object ob, string str)
```

The `tell_room()` efun sends a message `str` to all objects in the room `ob`. If the objects that receive the message `str` are not users, `str` will be intercepted by the local function `catch_tell()` to be manipulated and acted upon, assuming that `catch_tell()` is defined in them. `tell_room()` returns `1` if it was able to send `str` to `ob`, or `0` if `ob` could not be found.

Optionally, ob can be given as a full path of the room to which tell_room() should send the message str.

test_bit()

```
int test_bit(string str, int num)
```

The test_bit() efun returns 1 if bit num is set in string str. If bit num is not set in str, 0 is returned.

Each character contains six bits. This means that a value between 0 and 63 can be stored in one character, as 2^6 == 64. The starting character is the blank (" "), which has a value of 0. The first character in the string str is the one that has the lowest bits. Following is an example:

```
test_bit("_",5);
```

The preceding example returns 1 because _ stands for the number 63 and, therefore, the sixth bit is set.

NOTE *Bitfields* (a string consisting of bits that are set) are cryptic and ugly to look at, but they are perfect for storing a large amount of status information (status meaning like the variable type status, either off or on). You can store the equivalent of six different status variables in one character of a string using bitfield techniques! Bitfields, however, are not for the faint of heart and are *rarely* used in LPC simply because they are not easy to work with.

this_interactive()

```
object this_interactive()
```

this_interactive() is an efun that returns a pointer to the object that initiated current execution chain. If it was not a user that initiated the current execution chain, then 0 is returned. call_out() and heart_beat() are two ways in which an execution chain could be initiated without an interactive user having been responsible. In such a case, 0 would be returned.

this_interactive() should not be used as a replacement for this_player(). If you try this and a call to command() is executed, the value returned by this_player() will change, reflecting the current player, while the value returned by this_interactive() will not.

this_object()

```
object this_object()
```

this_object() is an efun that returns the object pointer of the current object.

this_player()

```
object this_player()
```

`this_player()` is an efun that returns the object pointer that represents the current player, that being the player who issued the command that initiated the current execution.

This function does *not* work in a `call_out()`. If you need to keep track of who `this_player()` was when a `call_out()` was initiated, then use a global variable that is declared as an object.

time()

```
int time()
```

The `time()` efun returns the number of seconds since January 1st, 1970, at 0.00 hours.

unique_array()

```
mixed unique_array(object *arr, string separator)
```

The `unique_array()` efun groups objects together for which the `separator` function returns the same value. `arr` must be an array of objects. If `arr` is not an array of objects, then it will be ignored. The `separator` is not, itself, a function call, but rather the name of the function upon which you want to base the order of your grouping. The function indicated by `separator` will be called only once in each object contained in `arr`. `unique_array()` returns an array of objects of the following form:

```
({
  ({Same1:1, Same1:2, Same1:3, ..., Same1:Y }),
  ({Same2:1, Same2:2, Same2:3, ..., Same2:Y }),
                      ...
  ({SameX:1, SameX:2, SameX:3, ..., SameX:Y }),
})
```

The following example returns an array of arrays, holding pointers to all player objects grouped together by their player levels:

```
mixed *objarr;
objarr = unique_array(users(), "query_level");
```

users()

```
object *users()
```

The `users()` efun returns an array of all interactive users.

write()

```
void write(string str)
```

The `write()` efun writes a message `str` to the current player (`this_player()`). The argument `str` also can be a number, which will be translated to a string. `1` is returned if write`()` was successful, and `0` if it failed.

write_bytes()

```
int write_bytes(string filename, int line, string text)
```

`write_bytes()` is an efun that writes the string `text` on the specified `line` in a given file `filename`. `filename` should be given as a full path and may not contain spaces. If the value of `line` is negative, `write_bytes()` writes `text` on line number `line` counted *backwards* from the end of the file. If the absolute value of `line` is greater than the number of lines in `filename`, then nothing is written and `0` is returned. `Write_bytes()` returns `1` for success and returns `0` if it fails during execution.

WARNING

`write_bytes()` overwrites data in the target file, unlike `write_file()`, which merely appends to the file.

write_file()

```
int write_file(string filename, string str)
```

`write_file()` is an efun that appends the string `str` to the file `filename`. `filename` should be given as a full path and may not contain spaces. `1` will be returned if `write_file()` was successful, and `0` will be returned if it fails.

Coding Rooms

Having seen comments, expressions, syntax, and the most common efuns and lfuns, you now should be ready to begin working on an area. The first step is to draw a simple map on a piece of graph paper so that you have some idea of what the area will look like. After you do this, you can begin to put the area's characteristics down into code.

As you know, you can use `inheritance` to incorporate all the aspects of one object into another. The primary use for `inheritance` is to allow wizards of all levels to inherit default objects and configure them for their own use. With the case of building rooms, the file `/room/room.c` usually is what is inherited to give the object you are working with all the aspects of a generic room. The `/room/room.c` file contains all the definitions for the way in which a generic room works, inclusive of necessary variables and functions, which you should use to configure your rooms. Table 13.7 shows the variables that you can use.

Table 13.7. Common global variables used to configure rooms.

Variable	*Intended Use*
`long_desc`	Holds the long description of a room, returned by `long()`. The `long_desc` variable is a string variable.
`short_desc`	Holds the short description of a room, returned by `short()`. The `short_desc` variable is a string variable.
`dest_dir`	`Dest_dir` is a string array of destinations followed by the direction that must be typed to go in that direction. `dest_dir` (short for `destination_direction`) is used to enable `add_action()` for the exits available in a room.

Following is an example of a complete, working room:

```
/*
 * Filename: /players/tarod/hell1/main.c
 * Code by Tarod@RealmsMUD
 */
inherit "room/room.c";

reset(arg)
{
  if ( arg ) return;   /* Saves some processing time */
  set_light( 1 );
  short_desc = "Avernus Main";
  long_desc = "You follow a path that twists and turns around "+
              "dead and rotting trees.\nAhead in the distance, "+
              "you catch a glimpse of the opening of a dark "+
              "cave.\nA shiver runs up your spine as a cold "+
              "breeze passes over your (now\nshivering) body.\n";
  dest_dir = ({
               "players/tarod/hell1/hell1.c", "south",
               "players/tarod/hell1/room0.c", "north"
             });
}
```

The preceding example is a simple room. The first thing it does is inherits `/room/room.c` to incorporate all the aspects of a room into itself. `reset()`then is defined so that the object is initialized when loaded. The first thing `reset()` does is check to see whether it was passed as an argument by the game driver. If `reset()` did receive an argument, the `reset()` defined previously terminates execution by returning. Why? Because the preceding information needs to be configured only on the first reset, which has no argument. Otherwise, it would be reconfigured every time `reset()` were called, wasting processing time.

The preceding room is marked as `lit` by a call to the efun `set_light()` with an argument of `1`. If this had not been done, the room would have been dark by default.

The short description, which is a string value that will be returned by `short()` in `/room/room.c`, is then set. Note that on most LPMUDs, the short description is what is seen when in brief mode. As a result, try to make your short descriptions informative and unique so that they can be used to differentiate rooms equally as well as long descriptions.

Next the long description is set. The long description is the string that will be returned when `long()` is called. Note the use of the preceding string addition, to keep the code clean-looking. This creates a little more work for the compiler when it loads an object, but as the compiler must only do it once, it is insignificant. Maintaining the readability of the code is more worthwhile than saving a few CPU cycles by eliminating the string addition in the preceding example.

Last, an array of destination files and direction commands is created. This array is set up by `add_action()` calls in `init()` of the room. You inherited `/room/room.c`, where `init()` has already been defined for you. The array, `dest_dir`, should be in the form of `({ filename, command, filename, command })`. The filename given should be expressed as a full path. `room.c` usually will support up to 10 exits, sometimes more. Once the preceding room is loaded, entering **`south`** while in the room will move you to `/players/tarod/hell1/hell1.c`, while entering **`north`** will take you to `/players/tarod/hell1/room0.c`.

The check for an argument in `reset()` need not be done on LPMUD version 3.0 or more recent versions that are running in native mode, as `create()` is used to initialize the object. Code for such an LPMUD might look like the following:

```
/*
 * Filename: /players/tarod/hell1/main.c
 * Code by Tarod@RealmsMUD
 */
inherit "room/room.c";

create()
{
  set_light( 1 );
  short_desc = "Avernus Main";
  long_desc = "You follow a path that twists and turns around "+
              "dead and rotting trees.\nAhead in the distance, "+
              "you catch a glimpse of the opening of a dark "+
              "cave.\nA shiver runs up your spine as a cold "+
              "breeze passes over your (now\nshivering) body.\n";
  dest_dir = ({
               "players/tarod/hell1/hell1.c", "south",
               "players/tarod/hell1/room0.c", "north"
             });
}
```

As you can see, the preceding two examples differ only in the name of the function used to initialize them, and by one `if()` evaluation.

Okay, so now you have one room. How do you link it to other rooms so that one may walk from one to the other? That's simple. You already defined the exits in your first room using `dest_dir`, but right now, those files don't exist because you have not yet written them. So, edit a file whose path matches one of the paths you defined as a valid exit in the room you just wrote. Then write the room as you normally would, setting `light` if needed, entering a short description by setting `short_desc`, entering a long description by setting `long_desc`, and setting up exits in `dest_dir`. The setting of `dest_dir` is what will link your rooms.

This next code sample is an example of a room that is linked to the previously discussed room (/players/tarod/hell1/main.c) by means of dest_dir.

```
/*
 * Filename: /players/tarod/hell1/room0.c
 * Code by Tarod@RealmsMUD
 */
 inherit "room/room.c";

 reset(arg)
 {
   if ( arg ) return;
   set_light( 1 );
   short_desc = "A damp, dark and dingy cave.";
   long_desc = "You enter a wide cave, which narrows and grows "+
               "darker to the east.\n";
   dest_dir = ({
                "players/tarod/hell1/main.c", "south",
                "players/tarod/hell1/room1.c", "east"
              });
 }
```

If you were standing in the room defined by previous examples and you entered north, you would be moved to this room. Here, dest_dir is defined to take you back to the room you just came from when south is entered. In this way, rooms are linked so that a user can move among them.

Now you want to have some monsters in the room and to put some things on your monsters. This is done by first writing and debugging any definition files of objects you want to use in your room, and then putting the code to create them into your room.

Following is an example of a room, that clones and moves a monster to itself on reset():

```
/*
* Filename: /players/tarod/hell1/room1.c
* Code by Tarod@RealmsMUD
*/
inherit "room/room.c";

reset(arg)
{
  object monster, money;

  if ( !present( "shadow" ) )  // If there's not a monster whose name is "shadow",
                               // we'll make one.
    {
    monster = clone_object( "players/tarod/monsters/shadow" );
    money = clone_object( "/obj/money" );
    money->set_money( random(2000) );
    move_object( money, monster );
    move_object( monster, this_object() );
    }

  if ( arg ) return;
  set_light( 1 );
  short_desc = "A damp and echoing chamber, filled with shadows";
  long_desc = "You step cautiously into a damp and shadow-"+
                "filled chamber.\nRough-hewn stone walls "+
                "disappear into inky blackness as\nyour eyes "+
                "scan their coarse contours.  Shadows flicker "+
                "in\nthe dim light of a lone torch.\n";
  dest_dir = ({
                 "players/tarod/hell1/room0","west",
                 "players/tarod/hell1/room2","northeast"
               });
}
```

This example works like other rooms. It differs from what you've seen so far in that two object variables, `monster` and `money`, are declared and then set to point at two objects cloned from the files `/players/tarod/monsters/shadow.c` and `/obj/money.c`, respectively. A `call_other()` to the function `set_money()` is made to the object pointed to by `money` in order to set the value of the money randomly within a range of `0` and `1999`. The object pointed to by `money` then is moved into the inventory of the object pointed to by `monster`. The object pointed to by `monster` is then moved to the inventory of `this_object()` (the room), with its inventory. After this, the set up of the room continues as normal. You should note that all of the `monster` and `money` configuration is done before the `if( arg )` check, so that at each call to `reset()`, including when the object is loaded, a `monster` with money on its person will be `cloned` and moved to the room if there isn't already one there.

Coding Monsters

Monsters keep players on the go for experience and weapons. Without monsters, there is generally little to do on a MUD unless the MUD is strictly a player-killing MUD or a chat-based MUD. So, you need to know how to code monsters.

Like rooms, monsters have a generic file that can be inherited and configured. The file `/obj/monster.c` should be inherited and then configured for use by you. Typically, a series of functions local to `/obj/monster.c` are used to configure a monster. This differs from configuring rooms, which uses variables, not functions.

Table 13.7 lists the standard functions you can use to configure a monster.

Table 13.7. Standard functions for configuring monsters.

function_name	*Intended Use*
`set_ac(num)`	Sets the monster's `armor` class (how well it avoids being hit) to the integer value of `num`.
`set_aggressive(arg)`	If `arg` is non-zero, makes the monster automatically attack someone who enters the room in which it is located.

continues

Table 13.7. continued

function_name	*Intended Use*
`set_al(num)`	Sets the monster's alignment to the integer value of `num`. Alignments work in ranges. The specifics of the range is MUD-dependent. However, commonly, the more evil the monster, the more negative the alignment value. The more pure/good the monster, the more positive the alignment value.
`set_hp(num)`	Sets the monster's hit points to the integer value of `num`.
`set_level(num)`	Sets the monster's level to the integer value of `num`.
`set_long(str)`	Sets the monster's long description to the string value of `str`.
`set_name(str)`	Sets the monster's name to the string value of `str`.
`set_race(str)`	Sets the monster's race to the string value of `str`.
`set_short(str)`	Sets the monster's short description to the string value of `str`.
`set_wc(num)`	Sets the monster's weapon class (how well it can hit things and how much damage it can do) to the integer value of `num`.

What follows is a complete, working example of the monster cloned by the sample room above (`/players/tarod/hell1/room1.c`).

```
/*
* Filename: /players/tarod/monsters/shadow.c
* Code by Tarod@RealmsMUD
*/
inherit "obj/monster.c";

reset(arg)
{
  ::reset( arg );        /* Makes sure the reset() function
                          * that was inherited gets called
                          * too.  This can be done with
                          * init() or any other function you
                          * need to define that already is
                          * defined in something you inherit.
                          */

  set_ac( 20 );
  set_aggressive( 1 );   /* Automatically initiates attack */
  set_al( -800 );        /* VERY EVIL */
  set_hp( 1100 );
  set_level( 20 );
  set_long( "Hovering on the edge of your vision, in the  "+
            "deepest recesses of darkness,\nthis creature "+
            "blends with the absence of light...and life.\n"
          );
```

```
    set_name( "shadow" );
    set_race( "demon" );
    set_short( "A dark, sinister shadow" );
    set_wc( 17 );
}
```

That's all there is to it! Assuming that your monster file loads properly, you can simply `clone` it and move it, within the `reset()` of your room, as you've already seen. The preceding example is a very simple monster. Monsters can have more options set in them than what you see here. As a player, you are aware that monsters sometimes talk, move, pick up things, and drop things. These monster capabilities are very MUD-specific. Each MUD does it a little (and sometimes a lot) differently. Thus, there is no catch-all example I can provide to show you how such things are commonly done. I strongly suggest that you work with simple monsters until you get a feel for it. Once you're comfortable with them, you might move on to monsters of increasing complexity and realism. Your sponsor, or an elder wizard, should be able to help you tackle the undertaking.

Coding Weapons

Weapons serve two purposes in the game. While players would like to think that the sole purpose for the existence of weapons is to provide them with something with which they can bash monsters (or players!), wizards generally think of weapons as something with which their monsters can bash players. When a weapon is wielded, the `weapon_class` (which represents how accurate the weapon is *and* how much damage it can add to its wielder's base damage) of the weapon is added to the player's base `weapon_class`. Not every monster has the physical attributes (such as claws, represented by a high `weapon_class` in the monster file) that make hand-to-hand combat easy. As a result, some monsters that you may create should have weapons. A normal man (created as a monster) with a `weapon_class` of `17` is ridiculous. It is more realistic to give said man a low `weapon_class` (such as `5`) and provide him with a sharp sword or a big ax that adds to his `weapon_class`. You must, of course, make the man wield the weapon you provide him with.

As with rooms and monsters, there is a generic weapon file, `/obj/weapon.c`, that should be inherited and configured to make a weapon. Also, as with monsters, the weapon should be cloned and then moved to a monster, much like the money you saw in a previous example. Once the weapon is in the inventory of the monster, you can make the monster wield it with a call to `command()`.

Table 13.8 lists the standard functions you can use to configure a weapon.

Table 13.8. Standard functions for configuring weapons.

`function_name`	*Intended Use*
`set_alias(str)`	Sets up an alias `str` to which the weapon will `id` in addition to its name.

continues

Table 13.8. continued

function_name	*Intended Use*
`set_alt_name(str)`	Sets up an alternate name `str` to which the weapon will `id` in addition to its name and alias (if any).
`set_class(num)`	Sets the weapon's weapon class (how much it adds to the capability of a living object to hit things and how much damage it can add) to the integer value of `num`.
`set_long(str)`	Sets the weapon's long description to the string value of `str`.
`set_name(str)`	Sets the weapon's name to the string value of `str`.
`set_short(str)`	Sets the weapon's short description to the string value of `str`.
`set_value(num)`	Sets the weapon's value (how much it's worth when sold to a shop) to the integer value of `num`.
`set_weight(num)`	Sets the weapon's weight to the integer value of `num`.

Following is an example of a complete, working weapon:

```
/*
* Filename: /players/tarod/weapons/dagger.c
* Coded by Tarod@RealmsMUD
*/
inherit "/obj/weapon.c";

reset(arg)
{
  ::reset( arg );        /* Makes sure the reset() function
                          * that was inherited gets called
                          * too.  This can be done with
                          * init() or any other function you
                          * need to define that already is
                          * defined in something you inherit.
                          */
  if ( arg ) return;
  set_alias( "old dagger" );
  set_alt_name( "an old dagger" );
  set_class( 5 );
  set_long( "The dagger is covered in rust, but still looks "+
            "functional.\n"
          );
  set_name( "dagger" );
  set_short( "An old dagger" );
  set_value( 200 );
  set_weight( 1 );
}
```

As you can see, configuring a generic weapon is a simple matter of providing the right information to the functions that exist for the purpose of setting up a generic weapon. Once you are sure your weapon works properly, you may want to give it to a monster. If you wanted the shadow (monster) you saw in a previous example to have this weapon, you would edit the definition file of the room which clones the shadow and add the following lines:

```
object weapon;
weapon = clone_object( "/players/tarod/weapon/dagger.c" );
move_object( weapon, monster );
command( "wield dagger", monster );
```

You would add the preceding lines just before the monster is moved into the room.

As a player, you are aware that there are weapons that do special types of damage, and have seen weapons that occasionally make special hits on targets of their own volition. Creating these types of weapons is MUD-specific—few MUDs do it the same way. Thus, I will not elaborate on the how-to's of it as there is no common example to which I can point and say "this is typical." If you want to create such weapons, you should ask your sponsor or an elder wizard how it is done on the MUD on which you are coding.

Coding Armor

Armor is used to protect its wearer from harm. If armor is worn, the `armor_class` (how protective the armor is) is added to the wearer's base `armor_class`. Not every monster has the physical attributes (such as a chitonous exoskeleton, represented by a high `armor_class` in the monster file) that make avoiding damage easy. As a result, some monsters that you make should have armor. A normal man (created as a monster) with an `armor_class` of `10` is ridiculous. It is more realistic to give said man a low `armor_class` (such as `1`) and provide him with a leather jerkin or a shield that adds to his `armor_class`. You must, of course, make the man wear the armor you provide him with.

As with rooms, monsters, and weapons, there is a generic armor file, `/obj/armor.c`, that should be inherited and configured to make armor. Also, as with monsters, the armor should be cloned and then moved to a monster, much like the money you saw in a previous example. Once the armor is in the inventory of the monster, you can make the monster wear it with a call to `command()`.

There are different types of armor that can be created and used. This represents different armor types, such as shields, helmets, gauntlets, rings, boots, suits of mail, and so on. Monsters and players can wear only one of each type of armor at a time. Different types of armor can be worn simultaneously, however. Table 13.9 lists standard types of armor.

Table 13.9. Standard armor types.

Armor Type	*Represents*
`amulet`	An enchanted amulet or a pendent worn about the neck.
`armor`	A suit of armor covering the torso and, possibly, the legs and/or arms.
`boots`	Covering for the feet, obviously.
`gloves`	Covering for the hands, obviously.
`helmet`	Covering for the head.
`shield`	A shield worn on one arm.

Table 13.10 lists the standard functions you can use to configure armor.

Table 13.10. Standard functions for configuring armor.

function_name	*Intended Use*
`set_ac(num)`	Sets the armor's `armor` class (how much it adds to the capability of a living object to avoid damage) to the integer value of `num`.
`set_alias(str)`	Sets up an alias `str` to which the armor will `id`, in addition to its name.
`set_long(str)`	Sets the armor's long description to the string value of `str`.
`set_name(str)`	Sets the armor's name to the string value of `str`.
`set_short(str)`	Sets the armor's short description to the string value of `str`.
`set_type(str)`	Sets the armor's type to one of the six available types: `amulet`, `armor` (or `armor` depending on how it's spelled on the MUD), `boots`, `gloves`, `helmet`, `shield`.
`set_value(num)`	Sets the armor's value (how much it's worth when sold to a shop) to the integer value of `num`.
`set_weight(num)`	Sets the armor's weight to the integer value of `num`.

Following is an example of a complete, working armor file:

```
/*
* Filename: /players/tarod/armor/leather.c
* Coded by Tarod@RealmsMUD
*/
inherit "/obj/armor.c"; // May need to be spelled armor
                         // on some MUDs.
reset(arg)
{
  ::reset( arg );        /* Makes sure the reset() function
                          * that was inherited gets called
                          * too.  This can be done with
                          * init() or any other function you
                          * need to define that already is
                          * defined in something you inherit.
                          */
  if ( arg ) return;
  set_ac( 2 );
  set_alias( "armor" );
  set_long( "While old and disheveled looking, you think that"+
            "this leather armor\nis better than none at all!\n"
          );
  set_name( "armor" );
  set_short( "Grimy, old leather armor" );
  set_type( "armor" );
  set_value( 200 );
  set_weight( 1 );
}
```

Once you are sure your armor works properly, you may want to give it to a monster. If you wanted the shadow (monster) you saw in a previous example to have this armor (an absurd notion, how does a shadow wear armor?), you would edit the definition file of the room that clones the shadow, and then add the following lines:

```
object armor;
armor = clone_object( "/players/tarod/armor/leather.c" );
move_object( armor, monster );
command( "wear armor", monster );
```

You should add the preceding lines just before the monster is moved into the room.

More sophisticated MUDs require armor to have a size. On such MUDs, a *large* man cannot wear *small* armor. This is a MUD-specific hack that few MUDs do the same way. Thus, I have no common example to give you and will let you ask your sponsor or an elder wizard to point you in the right direction if you need to deal with this.

Coding Containers

Players can only carry so much on their person. Gold, weapons, armor, and just about any other object they can pick up has weight. Containers help to organize a player's inventory and usually allow the player to carry a bit more. To make a container, a generic container file, `/obj/container.c`, should be inherited and configured.

Table 13.11 lists the standard functions you can use to configure containers.

Table 13.11. Standard functions for configuring containers.

function_name	*Intended Use*
`set_alias(str)`	Sets up an alias `str` to which the container will `id` in addition to its name.
`set_alt_name(str)`	Sets up an alternate name `str` to which the `container` will `id` in addition to its name and alias (if any).
`set_long(str)`	Sets the container's long description to the string value of `str`.
`set_max_weight(num)`	Sets the maximum weight the total weight of the container's contents can be. If putting an object into the container will cause this maximum to be exceeded, the player cannot put the object into the container and is told that the container is full.
`set_name(str)`	Sets the container's name to the string value of `str`.
`set_short(str)`	Sets the container's short description to the string value of `str`.

continues

Table 13.11. continued

function_name	*Intended Use*
`set_value(num)`	Sets the container's value (how much it's worth when sold to a shop) to the integer value of `num`.
`set_weight(num)`	Sets the container's weight to the integer value of `num`. This is how much the container weighs regardless of the total weight of its contents.

Following is an example of a standard, working container object.

```
/*
* Filename: /players/tarod/obj/bag.c
* Coded by Tarod@RealmsMUD
*/
inherit "/obj/container.c";

reset(arg)
{
  ::reset( arg );         /* Makes sure the reset() function
                           * that was inherited gets called
                           * too.  This can be done with
                           * init() or any other function you
                           * need to define that already is
                           * defined in something you inherit.
                           */
  if( arg ) return;
  set_alias( "bag" );
  set_alt_name( "bag" );
  set_max_weight( 21 );
  set_name( "large bag" );
  set_short( "A Large Bag" );
  set_long( "This is an extremely large bag, capable of "+
            "holding quite a bit.\n"
          );
  set_value(40);
  set_weight(2);
}
```

On MUDs that are programmed to deal with armor sizes, containers usually will have a size, as well. In such a case, a player will not be permitted to put a *large* object into a *small* or *medium* container. The code for this, as with armor size code, is MUD-specific and is not dealt with in this book. You should consult a high level wizard or your sponsor to learn how to deal with sizes.

Coding Treasure

Treasure objects are provided for two reasons: as a reward, similar to money, for undergoing an ordeal and, often enough, as something required for completion of a quest. Not all treasure is quest material. Most treasure is simply an object with `can_put_and_get()`, `drop()`, `get()`, `id()`, `long()`, `short()`, `query_name()`, and `query_value()` defined in it.

NOTE

`query_value()` is not always used. On some MUDs it may be called something else, but there *will* be a function that is MUD-specific that should return an integer value representative of how much an object is worth when sold to a shop. If the function is not `query_value()`, ask a high level wizard or your sponsor what the appropriate function is, or find and read the code to a shop to see what function it makes a `call_other()` to for determination of the value of an object that is being sold to it.

Following is an example of a treasure object. Note that no inheritance has been made.

```
/*
 * Filename: /players/bleys/treasure/diamond.c
 * Code by Bleys@AfterHours
 */
int worth;  // Holds the value of the diamond when it is
            // determined when this object is loaded.

reset(arg)
{
  if ( arg ) return;
  worth = random(5000)+1;
}

can_put_and_get() { return; }

drop() { return; }

get() { return 1; }

id(str) { return str == "diamond" ¦¦ str == "flawless diamond"; }

long()
{
  write( "You see a diamond that must be as large as your fist. "+
         "It appears to be\nflawless and must be worth a "+
         "fortune!\n"
       );
  return 1;
}

short() { return "A flawless diamond"; }

query_name() { return "diamond"; }

query_value() { return worth; }
```

Many MUDs do not have generic treasure objects to be inherited. As you have learned how to configure generic objects, I will leave it to you to see if there is a file `/obj/treasure.c` that can be inherited. If you find such a file, simply read through it, paying attention to the functions that begin with `set` and to what types of arguments they expect. Once you've done that, edit a file, inherit `/obj/treasure.c`, and write a `reset()` function in which the available `set` functions are used appropriately.

Attaching Your Area to the Main Map

By now, you have seen enough examples to create an area without much of a problem. In addition, you have been exposed to the grammar (comments, operators, efuns, and lfuns) and syntax of LPC. You should be ready to create an area using this work as a reference for most, but not all, of your questions.

When you have completed your area, you should have another wizard of any level (high level wizards are best—but it's hard to get their attention) look through the area and critique it. Accept the criticism and then go back and modify your area according to the suggestions made. The wizard should be looking for typing errors in the area's descriptions, as well as possible flaws or bugs in the code.

Once the area is clear of bugs, typing errors, and any suggestions you want have been implemented, you should talk to your sponsor or an elder wizard about attaching your area to the main map and opening it for use by players. A target date should be set and an announcement on a board (or repeated shouts, though this is annoying) should be made so that players know something new is coming online. It is up to you whether you disclose the area's location or keep it to yourself so that players can just happen upon it.

Try to avoid choosing a place on the game map that doesn't fit the style of the area you have built. Placing an area that is a grassy knoll into the middle of a village of 10,000 is not good. Try to keep in mind the feel of the location to which you are attaching.

Watching for Undocumented Features

When you write objects, you need to keep in mind the mentality that the players in the game usually have, that being, most want something for nothing. While most LPMUDs penalize players who cheat, there is a fine line between cheating and utilizing an "undocumented feature," a.k.a., a bug.

To illustrate the kind of thinking I'm talking about, you should know that early LPMUDs were riddled with bugs. In the early 1.3.3 versions, a player could `give -100 coins to John`, and assuming that John was in the room with the player, John would lose 100 coins and the player who gave him negative coinage would gain 100 coins.

Similarly, in my days as a player, I encountered armor that I could purchase from an armory at 50 coins and take just down the street to a shop and sell for 200 coins. Obviously, I spent some time running back and forth from one to the other and kept what I knew to myself so that word would not spread and the *feature* would stay a bit longer.

Give all logical possibilities some thought when you write code. If you simply write it and put it into play without giving it a second thought, you are likely to miss something that some clever player might be able to anticipate you missing. The result could be ugly.

An example of this would be linkdead players that turn into statues. Some MUDs turn a player into a statue when linkdeath (disconnection without quitting) occurs. All the player's items are left on the statue, but the statue can be picked up. There have been instances of players being sold to shops for quick cash (at which point they end up in the *back room* of the shop and can get at everything that is in the shop). This means the player who sold the statue gets a small amount of cash, and the player who reconnects finds himself sitting in the midst of a hoard of useful things.

On that same note, many LPMUDs have a `trash bin`—some object that destroys objects that can't be sold to a shop. There have been instances of players throwing statues into these bins so that unique (one of a kind) equipment that is on the linkdead player will be put back into game play in 45 or so minutes.

Think through your code. Watch for undocumented features.

Stones Left Unturned

As every LPMUD is different, obviously I cannot cover every possibility. I deliberately chose to avoid addressing some things that are common to most LPMUDs for one reason or another. Here, I explain why.

Mappings

I do not address the variable type `mapping` or any efuns that pertain to mappings. Mappings are similar to arrays. Frankly, anything you can do with mappings you can do with arrays. Certain people within the core of the MUD world have argued that mappings are more efficient than arrays. Others have tested the theory and found mappings to be *less* efficient than arrays. As the efficiency advantage is debatable and mappings are redundant anyway, I opted to omit them from this chapter. If you want to learn about mappings, you should try to find documentation on declaring them as well as documentation on the following efuns: `mappingp()`, `mkmapping()`, `m_delete()`, `m_indices()`, `m_sizeof()`, `m_values()`

euids

Three efuns, `export_uid()`, `getuid()`, and `seteuid()`, are used to handle security-intensive code (such as code that needs special permissions to read, write, or do otherwise restricted things). Don't worry about UID security for the most part if you are using generically inherited objects. By the time you are working with custom objects, you should be a high level wizard and know how security works on the MUD on which you code (every MUD's security differs to some extent). Due to the wide variance in MUD security code, I have elected to leave UID security unaddressed. LPMUD versions 2.4.5 or older don't have UID security anyway. Versions 3.0 or more recent have it and generally use it. LPMUDs running in compat mode have it but often elect not to bother with it.

floats

I do not address the variable type `float` or any efuns that pertain to it. `floats` are variable type that can have a decimal values. While `floats` are useful in C or C++ coding, I have yet to find a use for floats in LPC that is worth the time expenditure made to enable LPC to support floats.

parse_command()

`parse_command()` is an efun that is used to parse strings. Almost no one uses it. `sscanf()` is much cleaner and easier to use. I suggest using `sscanf()` rather than wasting time with `parse_command()`.

tracing Functions

The efuns `trace()` and `traceprefix()` are almost never used. Both are cumbersome, CPU-eating debugging tools that next to no one bothers with. Thus, they also have been omitted from this chapter.

Summary

In this chapter, you have been exposed to the essentials of being a wizard. You know what the common, essential wizard commands are, and you have seen how they are used. You know what a MUDlib is and where, in general, things are located within it. Lastly, you have been exposed to the code that wizards use to create the "magic" within the game. Using the tools you now are aware of, and the knowledge you've recently gained, you should be ready to build a castle using LPC on a MUD where you have obtained the title and responsibilities of wizard.

14
CHAPTER

Programming MOOs

By Chris Stacy

The *virtual reality* of the MOO is created by programmers, who create and program the *objects* that make up the MOO. Everything in the MOO, including characters, the rooms they inhabit, and the things they pick up and play with, are objects.

This chapter teaches you how to create new objects, modify existing objects, and program the objects to do what you want.

If you're not already a programmer in some other language, don't worry. LambdaMOO has been designed to be easy to learn to program.

The Elements of MOO

MOO stands for *MUD Object Oriented*, which tells you that the way the system works is oriented around *objects*. All the objects in the MOO interact with each other to create the virtual reality. Each object is programmed to do certain things, such as be picked up or moved around, be looked at, and so on. The human characters who are playing on the MOO are represented by `player` objects, which are programmed to (upon command from the player) pick up things, `say` things to other players, move around, and so on.

Every object is owned by some other object (generally a player). Player objects are generally owned by themselves.

Object Numbers

Every object has a number, called it's *object number*, or *objnum* for short. When referring to an object by its number, you use the pound sign (`#`), followed by the digits. So, object number eleven is written as `#11`. The MOO automatically assigns each new object its own number (you don't get to pick them). An object's number never changes—you can refer to any object in the whole MOO by its number.

TIP

Objnum is pronounced pretty much like it looks: *ahh-b-j-num*.

Verbs and Properties

Every object has some *verbs*. Each verb is a unit of behavior, a little program that specifies one way that the object interacts with the other objects on the MOO. To make objects behave the way you want, you define and program verbs into them.

A verb has a name, hopefully something that brings to mind what it does. For example, a particular verb on a bouncy ball might be called `bounce`.

NOTE

Sometimes in discussions people write the name of the verb preceded by a colon (`:`), as in `:bounce`. In the examples here, it is just written as `bounce` (unless it's in a place where you are really supposed to enter the colon).

Every object has properties. A *property* is a slot in the object that remembers some piece of information. Every property has a name. For example, an object that represents a room might have a property called `dark` that remembers whether it happens to be dark inside the room. A room also might have a property called `exits` that contains information about ways to exit the room. The piece of information that a property remembers is called the property's *value*. Collectively, objects remember all the states of the virtual world in their properties.

In programs, properties are referenced by affixing a period (`.`) to the front of the property name. For this reason, when speaking aloud, programmers often pronounce a property name as "dot *something*." For example, "I'm not sure what dot dark on rooms is for, but I read somewhere that it has something to do with whether you can see anything when you're inside the room."

Object Classes

The MOO is made up of objects of all kinds. Every object is patterned after some other kind of object, called its *parent*. An object has only one parent. When you create an object, you specify which object is the parent, and the MOO creates a new object that's like a copy of the parent, except that it has its own object number. A parent object can have as many children objects as needed or necessary.

The new object (the *child*) has the same properties as its parent, and it also has the same verbs. So, at least at first, it looks and acts just like its parent. You might want your child object merely to be a copy of its parent object; or, you can make new verbs and properties on the child object, specializing it to make it more interesting than its parent.

Each object has a chain of ancestors: its parent, its parent's parent, and so on. An object's *class* is the type of object it is—which branch of the family it's from.

Suppose that there is an object that represents a bottle of floor wax. You could use the `bottle of floor wax` object as a parent, and create a new object called `Shimmer`. When talking about the ancestry of the `Shimmer` object, you would say that "Shimmer *is of class* bottle of floor wax." Further suppose that you then took `Shimmer` as a parent, and made new object called `Improved Shimmer`. The new `Improved Shimmer` is a member of the `bottle of floor wax` family, and also a member of the `Shimmer` family, because those are both ancestors. You could say that `Improved Shimmer` was a member of either of those classes, depending on how specific or how general you wanted to be.

MOO is a *single-inheritance* object system, which is a fancy way of saying that every object has exactly one parent, and is only related to other objects along one branch. For example, a particular object cannot simultaneously be a room and a player, nor can it be both a floor wax and a dessert topping.

The Database

The *database* is comprised of all the objects in the MOO. The original objects that existed when the MOO was started for the very first time (before all the characters, rooms, and other things are created) are called the *core* database objects.

Your Client

The program that you use to connect to the MOO is called your *client* (for more information about clients, see Chapter 10). The most simple client is a program called *telnet*, available on most computers. However, telnet really doesn't do anything for you besides provide a raw sort of connection. In fact, if you're just using telnet, people will say that you don't have a client, and they will advise you to get one.

Without a client (better than telnet), nothing prevents the text you type and the messages from the MOO from coming out all at once and getting mixed together on your terminal. It won't be mixed up as far as the MOO can tell, but it will appear jumbled on your screen, and will be very difficult for you to keep track of. If you're going to program the MOO, it's highly recommended that you use a good client.

The Server

The *server* is the computer program underlying the MOO. It maintains the Database, keeping track of all the objects. The server takes each command sentence that you type, figures out what part of the database it applies to, and invokes the appropriate objects, causing their verbs to execute by interpreting the MOO programming language.

You don't program the server; it's not written in the MOO programming language, and you can't change the way it works with MOO programs. Rather, the server is what makes MOO programs (verbs) go!

To use the MOO, you connect to the server with your client, and send the server commands, such as `get rock`. The server parses your commands, figuring out which verb you're using and the objects to which you are referring, and invokes your commands upon those objects. The objects, in turn, may direct the server to send you back some kind of message, such as `You pick up the rock`.

Basic Objects

The MOO starts out with few very basic objects, and then programmers come along and create more objects that are like those, but adding new properties and verbs, and giving the MOO its own unique feel. If every object on the MOO is the child of some other object, where does it all begin? The answer is an object called the `Root Class`, which has no parent, and from which all other objects are descended.

Most things that you can pick up and do things with are "generic things." All rooms are based on the *generic room*, and player objects are, of course, based on *generic player*. The *generic container* is just like a *generic thing*, except that it understands the idea of containing other things (that is, it has verbs for putting things inside it, looking inside it, and taking them out).

The `Root Class` object is object `#1`, and the other basic generics have low numbers, too. Rather than memorizing their numbers, however, you can refer to them with the following special notations:

`$root_class`	The Root Class
`$player`	Generic player
`$room`	Generic room

`$thing`	Generic thing
`$container`	Generic container

If you're speaking aloud to someone, you would pronounce the dollar sign and say, "dollar room" to refer to `$room`.

The word "generic" indicates that these objects exist only for the purpose of having children, and are not to be used directly. For example, you would never walk into the `generic room`. Instead, you would make children of it to be rooms that players will move around in. "Generic" indicates an abstract concept, rather than a specific instance.

Player Classes and Programmer Bits

When you connect to the MOO, you interact with its virtual world of objects through your player object. Your player class determines many of the capabilities that you will have. For example, the most basic player class, `$player`, does not have the verbs needed to create new objects.

The `$builder` class has commands for creating new objects, but lacks the verbs needed to write programs. If you just want to `build` objects (but not invent new kinds of objects), then you just have to be a `$builder`.

To be a MOO programmer, you will need to be a `$prog`, or some class descended from there. To become a programmer, contact one of the wizards on your MOO. In one wizardly action, they will transform you into a programmer: your programmer bit will be turned on, and your player class will automatically be adjusted to be a descendant of `$prog`, if needed.

About Player Classes

On most MOOs, there are a variety of player classes to choose from, and you'll have to ask around to find out which one your player should be. Player classes determine, to some extent, how you appear to look and act to other players, and certain ways that other things behave towards you. On some elaborate MOOs, there are lots of personal taste choices to be made in this area. Note that even though `player` object owns itself, some other `player` owns your `player` class (your `parent`). This gives that person some control of your player, so you should only switch to a `player` class owned by someone you trust. You change your `player` class by using the `@chparent` command, described later in this chapter.

About Programmer Bits

In addition to being a `$prog` object, you also will need a *programmer bit*. Every player object on the MOO has a property called `programmer`, which records whether you have a programmer bit. The property holds the answer to the yes-or-no question, "Is this player allowed to write programs?" When an attempt is made to write a program, the server checks your `programmer` property to see if you have the right to do that. If the answer is `false`, programming will be denied, but if `programmer` contains a `true` value, then the server will allow you to program. Only wizards can turn programmer bits on and off, using their special `@programmer` command.

Before You Begin Programming

You're going to want to find a quiet virtual room on the MOO where you can work without a lot of spam. If you already have your own room, go there. Otherwise, there's probably some fairly quiet room nearby. Explore a little and look for someplace that's not too busy with other players talking and playing.

Creating and Customizing Basic Objects

Most MOOs have a rich collection of many different types of objects. There usually are many children of the most useful and popular objects. As a programmer, you can add new verbs and new properties to a child, thus making the child into a new kind of object. But if you don't need new types of behavior from the object, you can simply change the values of existing properties, such as the `name` and `description`.

To create an object, you first have to decide which class of object you have in mind. Suppose that you want to make a ball to bounce around. Like most things, the appropriate parent would be `$thing`.

```
@create $thing named ball
```

```
You now have ball with object number #2361 and parent generic thing (#5).
```

Note the object number that comes back (`#2361` in this case).

On your MOO, the preceding command example will get you some other objnum than `#2361`, so in this example, you'll have to remember to substitute the real numbers. Except for the few basic objects that exist on every MOO (such as `#1`), there's no way to predict what the object numbers will be.

```
@create parent class named name

@create parent class named name,name,name,...
```

The `@create` command creates new objects. You specify who the parent should be, and what the resulting object should be named. The server responds by creating the object, putting it in your inventory (the list of things you are `holding in your hand`), assigning the new object an objmun, and telling you about it. (By the way, there's no limit to the number of things you can hold in your inventory.)

The `inventory` command (which can be abbreviated `inv`) tells you which things you're holding.

```
inventory

Carrying:
ball
```

Every object has a *name*, which is just something convenient by which you can call it. The preceding object is named `ball`. But there might be more than one ball in the whole MOO, and they could all be named `ball`, so how can you tell them apart? The answer is that while object names are *not* unique, objnums *are*. You can't tell them apart by name, but they will always have different objnums. When you command the server to do something with the `ball`, it will try to find an object with that name. If you're only holding one `ball`, or if there's only one `ball` in the same room as you, that's the one the server will pick.

Aliases

An object can have multiple names, called *aliases*. If you wanted to be able to call the object either `ball` or `fun ball`, you could have said:

```
@create $thing named ball,fun ball
```

This could be a little tricky, though. Suppose that you issued the command once with just `ball`, and then once as in the preceding code with the multiple names. Then you would be holding two objects, both named `ball`, but one of them is also known as `fun ball`.

If you just say, for example, `drop ball`, the server will complain:

```
I don't know which "ball" you mean.
```

However, `drop fun ball` would work. Now there's a `fun ball` in the same room with you, and you're still holding the other `ball`. If you say `drop ball` at this point, the server would pick the ball you're holding, because it's closer than the one that's on the virtual floor.

In general, having two objects of the same name in the same room at the same time is inconvenient, and you'll find yourself resorting to objnums in such circumstances.

Object Numbers Always Work

You can only refer to objects by their name if they're in the same room as you (or if you're holding them). But you can always refer to any object that's anywhere on the entire MOO by its objnum, as in the following:

```
drop #2361

Dropped.
```

Renaming Objects

The names of your objects are just for your convenience and you can change them anytime you want using the `rename` command:

```
@rename obj to name
```

```
@rename obj to name,alias,alias,alias,....
```

```
@rename obj to :alias,alias,alias,....
```

The `rename` command renames an object to a new name, optionally including some aliases, or can also (as in the preceding third form) be used to just set the aliases without changing the object's preferred name. The `obj` part of this command (or any command) can be either a name (`ball`) or an objnum (`#2361`), whichever is more convenient for you.

Object names can be any combination of letters and numbers and spaces, but it's best to stick to single words, or maybe two or three descriptive words. Avoid numbers; they're just confusing. If you give your object a long name, it's sometimes useful to also give it a short alias.

If you had an object representing a bottle of soda pop, you might want to be able to refer to it in a variety of ways, so you could give it several aliases, as in the following:

```
@rename bottle to bottle, soda bottle, returnable soda bottle, returnable
```

Player Names Are Unique

Your player object has a name, and that's your name on the MOO. Unlike regular objects, you can only change your name to something which no other player has selected for themselves.

If you try to rename a player object to a name that's already taken, the MOO will complain. (See the section entitled "Ownership and Permissions," later in this chapter, if you're curious about the technical details of how this restriction is enforced.)

For the purpose of this chapter, your name is `Reader` and your player object number is `#1007`.

Describing Your Objects

You should give a description to every object that you own, so that when people `look` at it, they see something interesting.

```
look ball
```

```
You see nothing interesting.
```

The `@describe` command gives a description to an object. Following is its syntax:

```
@describe obj as "text
```

The *text* can be anything you want. You should capitalize and punctuate the text the way you want it to appear. You usually will want it to read like an ordinary sentence, perhaps without the verb. There is no closing quotation mark at the end of the text.

```
@describe ball as "A bouncy rubber ball.
```

```
Description set.
```

```
look ball

A bouncy rubber ball.
```

What Things Can $things Do?

Children of `$thing`, the *generic* `thing`, don't do very much by themselves. You can name them, describe them, pick them up and drop them, and that's about it. We'll come back to the bouncy rubber ball, `#2361`, a little later in this chapter.

Player Descriptions

The `$player` object (and therefore, your player object, since it is a descendant of player) has several properties that control how you appear to other players. If you haven't already done so, you should set them now. The most important one is, of course, your `description` property, set with the `@describe` verb. You can refer to your own player object as `me`.

```
@describe me as "A pleasant person with good taste in books.
```

You also should set your gender, which arranges for your pronouns to come out correctly.

```
@gender female

Gender set to female.
Your pronouns: she,her,her,hers,herself,She,Her,Her,Hers,Herself

look me

Reader
A pleasant person with good taste in books.
She is awake and looks alert.
Carrying:
ball
```

The basic genders include `female`, `male`, `either`, `Spivak`, `splat`, `plural`, `egotistical`, `royal`, and `2nd`. Some MOOs have defined other genders. If you're not sure which gender you're currently registered as, or you want to see a list of possible genders, just type **`@gender`**, and the MOO will tell you. Political correctness aside, some of the genders are pretty weird; my favorite is Spivak.

Changing Parents

When an object is created, it inherits the properties and verbs of its parent. (Its parent inherited some of its properties and verbs from *its* parent, and so on. Each parent along the way contributes or changes some properties and verbs.) When you chose the parent for your object with `@create`, you chose a particular set of inherited verbs and properties, giving your object the same behavior as its parent.

Later, you may have decided that your object should behave in a different way—the way that some other object works. You can change your object from the class it is in now, to another class, giving it a new parent and forgetting about the old parent.

This usually only makes sense if the new parent is very similar to the old parent. For example, if the new parent is just down some other branch of the same *family tree*.

Example: Changing Your Player Class

All player objects are descendants of `$player`, but there might be many diverging types of player classes. Changing your player class makes sense because they all have the essential necessary verbs and properties for you to function as a player. Of course, you will lose whatever verbs and properties you inherited from your old parent (unless your parent was inheriting it from some ancestor in common with your new parent.)

The following example switches from the player class you are now to the class of players descended from an object named `Revenant Player Class`:

```
@chparent me to #5409
```

```
Parent changed.
```

The server recognizes `me` as referring to your very own player object.

Unless the object (in this case, the parent) is in the same room with you, which may be unlikely, you will have to refer to it by object number. How do you know the object number of the `Revenant Player Class`, for example? How could you know that such a class even existed in the first place? Generally speaking, you will have to just ask around on your MOO.

Suppose that you see another player (in the same room as you), and you would like to be the same player class as they are. You can find out what their player class is with the `@parents` command, as in the following:

```
@parents Fred
```

```
Fred(#59845) Revenant Player Class(#5409)
Sick's Sick Player Class(#49900)
Detailed Player Class(#6669)
Generic Super_Huh Player(#26026)
Politically Correct Featureful Player Class(#33337)
Player Class that does substitutions and assorted stuff(#8855)
Generic Player Class With Additional Features of Dubious Utility(#7069)
generic programmer(#217)
generic builder(#630)
generic player(#6)
RootClass(#1)
```

`@parents` gives a list that shows the name and object number of the object you specified (in this case, a player named `Fred`), and the name and object number of all of its ancestors, all the way back to the `Root Class`.

Getting Rid of Objects

When you have an object that have finished playing with and that you wish didn't exist anymore, you can `recycle` the object. Recycling makes the object go away, permanently; it erases the object from the MOO.

```
@recycle object
```

You can `recycle` any object that you own. When you recycle an object, its object number eventually becomes available for reassignment to some other new object. (That's why it's called `recycling`, rather than `destroying`.)

Delayed Recycling

On some MOOs, when you recycle an object, it's sometimes possible to get it back, if you act quickly enough. Most MOOs don't work that way, though. Generally, once you have recycled an object, it's gone for good.

Sentences, Verbs, and Objects

Understanding verbs and objects is what MOO programming is all about. Look at how the server parses your command sentences into verbs and objects, and decides which verbs to execute. This probably is the most complicated part of MOO programming.

Your command input consists of a simple sentence, except that you don't use articles (`a`, `an`, `the`), and there's no ending punctuation. Your sentence is broken up into words; the server understands sentences of the following forms:

```
verb
verb    direct-object
verb    direct-object    preposition    indirect-object
```

The prepositions that the server understands include `with`, `using`, `in`, `on top of`, `through`, `behind`, `for`, and `is`.

Verbs need to know which objects they should affect to do their work. The server assists by trying to provide the objnums of the direct and indirect objects in the sentence, if it can figure them out. (If it can't, the verb just winds up getting the literal words from the sentence; the verb itself has to figure out what to do then.)

The direct and indirect-objects in your sentence can be written either as objnums (such as `#2361`), or object names (such as `ball`). You could say

```
get #2361
```

or

```
get ball
```

The server tries to match up the objects in your sentence with objects in the MOO. Usually people manipulate objects that are nearby that they can `see`, and they usually call the object by its name.

NOTE

Objnums are only needed when there are nearby things with the same name, or when you want to refer to something that is far away.

When you use a name rather than an objnum, the server goes looking for objects that you are holding or that are in the same room as you. Abbreviations of object names are allowed, and the server will try to figure out what you mean. For example, if there's only one object named `basketball`, you may be able to refer to it as just `bask`.

The server also knows that the word `me` refers to your own player object, and the word `here` refers to the room that you currently are in.

Verbs

Verbs are part of objects, and every verb has a name, such as `say`, `bounce`, or `drop`. Different objects can have verbs of the same names that do different things. The server has to figure out which is the correct verb to invoke.

The server assumes that the first word of your sentence is the verb. You also can begin your input with one of the following special characters, which are replaced by the indicated verb.

`"`	`say`
`:`	`emote`
`;`	`eval`

NOTE

A space is automatically included after the substitution, just as if you had typed the word.

The server searches for the named verb by looking at your player object, the room you're in, the direct-object, and the indirect-object, in that order.

If name of the verb is `laugh`, for example, you can think of the server as asking the following questions:

- Does this player know how to `laugh`?
- Does this room have a way that players in it should `laugh`?
- Does the direct-object know how to `laugh`?
- Does the indirect-object know how to `laugh`?

Arguments

In computer lingo, an *argument* is a piece of information that a program requires at the time it is invoked. A program is written to expect certain arguments, and doesn't work if they are not provided when you try to invoke it. *Argument* also is used to mean the placeholder for the expected information. For example, if someone were to say, "The program takes a date and an amount as arguments," they would mean "When you invoke the program, you must give it a date and an amount." Such a program would be said to "take two arguments."

On the MOO, verbs that process sentences can take several arguments, including: the direct-object, preposition, and indirect object. As you will find out, the *argument specifiers* for the verb define which of those arguments the verb takes, and also puts some restrictions on what the arguments can be.

Argument Specifiers

Not every verb is applicable to every sentence. Each verb has *argument specifiers* that tell the server which kind of sentences it will work with. The argument specifiers are a template indicating whether the verb handles direct or indirect objects and prepositions.

When the server is searching for a verb with the right name on the player, room, and sentence objects, it also checks each verb's argument specifiers. The verb will only be selected if both the name and the argument specifiers match the sentence.

Every verb always has three argument specifiers, always written in the order: direct-object, preposition, indirect object. The specifier is just a special word that says whether the part of speech is applicable to this verb.

For prepositions, the specifier is either a word that begins a prepositional phrase (such as `with`), or the word `any`, if just any old kind of preposition is allowed. It also can be the word `none` if no preposition is allowed for this verb.

For the direct-object and indirect-object objects, the specifier is the word `any`, `none`, or `this`. The word `any` means that any object can be used as that part of the sentence, and `none` means that the corresponding part of speech must not be present. The word `this` means that the verb will only work on the very object that the verb itself is defined on.

Suppose that there is a `sword` object that has a verb named `slash`. It might have the following argument specifiers:

```
slash     any with this
```

meaning that any object can be slashed with this sword. Sentences such as `slash Sam with sword` could match this verb.

Another example might be a `bottle`, perhaps one that has been labeled cryptically: `DRINK ME`. The bottle might have a verb named `drink`, with the following argument specifiers:

```
drink     this none none
```

meaning that only sentences such as `drink bottle` would apply.

The server searches through the player, room, direct, and indirect objects, in the order described previously, looking for a verb that has both the right name and the right argument specifiers. When the server finds a verb that matches, it invokes that verb and gives it all the information about the sentence. The verb is informed as to all the words that you typed in the sentence, and also which objects (if any) the server located that match the names in the sentence. Invoking the verb activates the object to do whatever the object's verb is programmed to do.

Say, Can You See?

The most popular command on any MOO certainly is `say`, which prints out your words to everyone else who's in the room, and is how groups of people chat with each other. How does it work?

```
"Hello, virtual world!
```

The server substitutes the word `say` for the `"` character, yielding the sentence:

```
say Hello, virtual world!
```

The verb name is `say`. There's no preposition there, so the direct-object is taken to be the whole string of words after the verb, `Hello, virtual world!` (the server looks for an object with that long and rather unlikely name, but finds none).

Now the server looks for a `say` verb of the proper form. First it checks your player object, but it doesn't find a `say` verb there. Next it checks the room you are in. Rooms are, in fact, where `say` is implemented. The room's `I` verb has the following argument specifiers:

```
say     any any any
```

meaning that absolutely any sentence would be acceptable.

Therefore, the server invokes the room's `say` verb.

The `say` verbs on rooms usually are programmed to take all the words in the sentence you typed (except for the `say`, of course) and print out `You say` *`whatever`* to the player who said it. It also prints out the message to any other player in the same room.

Emoting (which is covered in Chapter 5) also is programmed on rooms very much like `say`.

Do You *Get* It?

Many objects have verbs that only work on themselves. The argument specifier for these verbs looks like:

```
this none none
```

Only sentences where the direct-object is `this` object will work. In the following example, no preposition or indirect-object is allowed.

```
get ball
You take ball.
```

In this case, the verb is named `get`, and the server finds an object named `ball` in the same room as you, so the direct-object will be `#2361`. The server now searches for the appropriate verb to invoke.

1. As always, first it checks your own player object. There is indeed a `get this none none` verb defined on your player object. But that's not the right verb to invoke, because the direct-object in the sentence was the `ball` (`#2361`). This verb only works if the direct-object is your player object.
2. Next, the server tries the room you're in, but there is no `get` verb defined on your room.
3. The server checks the direct-object to see if it has a verb named `get`. All `$things`, including your ball, have a `get` verb, as in `get this none none`.

   Those argument specifiers require that the direct-object be this very object. The direct-object in your sentence is indeed this very ball, so that's a match!

The server invokes the ball's `get` verb, which arranges for the `ball` to be in your possession and prints out an appropriate message, such as `You get ball` or `You already have that!`.

Verbs on Player Objects

Usually, when you command some action (such as `get`) towards an (direct or indirect) object, such as the ball, that object's own verb is the one that is invoked. This isn't always so, however.

As explained previously, the `player` object is the first thing that the server looks at when searching around for a suitable verb. The types of verbs you find on `player` objects tends to fall into two categories. Some verbs simply modify the `player` object itself (perhaps changing the appearance), or are utility commands for the player. Other `player` object verbs might manipulate other objects (in some way that's unique to the player class). The following three sections look at examples each kind.

@sethome

The `@sethome` verb is used to change your home location to the room that you are now in. If you were in a `room` called `73 Mystery Avenue`, it would be used like this:

```
@sethome
73 Mystery Avenue is your new home.
```

The server sees that the verb is named `@sethome`, and that there are no objects to parse. It looks for an appropriate verb, first on your player object, and finds one:

```
@sethome     none none none
```

Those argument specifiers mean that the verb is applicable only for "bare" sentences with no objects.

That's the kind of sentence you typed, so the server invokes your player object's `@sethome` verb. The verb adjusts your player object's `home` property and prints a message to that effect.

@gender

In the following sample, the name of the verb is `@gender`, and the direct-object is `spivak`.

```
@gender spivak
Gender set to Spivak.
```

The server first looks on your `player` object for an `@gender` verb, and finds one specified as

```
@gender     any none none
```

which means that anything can be the direct-object, and no preposition or indirect-object is permitted.

The server invokes your `@gender` verb. As it happens, the `@gender` verb is programmed not to care that there is no such object as `spivak`. The verb is only interested in the literal word "spivak," and it sets your gender as indicated.

If you enter

```
@gender
```

by itself, the procedure is much the same. This time, there's no direct-object in the sentence, but that's okay. The `any` argument specifier means not only that any object will be allowed for that part of speech, there doesn't even need to be a word for that part. (This is different from `none`, which means that there may not be such a part of speech under any circumstances.)

In the case of no direct-object, the `@gender` verb is programmed to print out what your gender is, and also supply you with a list of all the possible genders.

@describe

The `@describe` verb, which is on your `player` object, is designed to enable you to modify the description of any object. (It can modify the description of any object, not just that of your own `player` object, or any other objects.)

```
@describe lump as "A sooty lump of coal
```

The verb is `@describe`, the direct-object is something called `lump`, the preposition is `as`, and the indirect object is the word string `A sooty lump of coal`. The server finds the following verb on your player class:

```
@describe any as any
```

The `lump` object matches the direct-object `any`, `as` matches the preposition, and your descriptive words all become the indirect object.

The server invokes the player object's `@describe` verb, which is programmed to arrange for the `lump`'s `description` property to be set to `A sooty lump of coal`.

Hey, *Look* Me Over

Another all-time biggie command is `look`, which prints out a description of a part of the MOO for you.

Alone, `look` gives a description of the room that you're in. The server parses the verb name, `look`, and because there are no other words, it doesn't need to parse any objects and so it begins searching for a verb.

Like the `say` verb, `look` is implemented by a room. The server invokes the room object's `look` verb, and the verb prints out a description of the room.

```
look ball
```

When you ask to `look` at some particular object, the server again parses the verb name, and then finds an object matching the name `ball`. So this time your `ball` (`#2361`) becomes the direct-object. To complete its job, the server goes looking for the appropriate `look` verb to run. As always, it checks the player first, and then it checks the room.

What happens next is somewhat peculiar. Suppose that the argument specifiers for the room's `look` verb were:

```
look      none none none
```

The new sentence, with it's direct-object `ball`, does not match that, so the server would not use the room's `look` verb. Next, the server tries the direct-object—the `ball`—and perhaps finds a `look` verb with these argument specifiers:

```
look      this none none
```

The `ball`'s own `look` verb would be invoked to print out a description of the ball. If you guessed that this is how it works, you have correctly understood how verbs work, and it's a good guess. However, in this case you would be *wrong*; the story is a little more twisted than that.

The `look` verb on the room actually has these argument specifiers:

```
look      any any any
```

This sort of short-circuits what you may have thought was the most obvious way of implementing `look`! The server will successfully match any `look` sentence to the room's verb.

The server invokes the room's `look`, with `ball` as the direct-object. The room's `look` verb then proceeds to `look` up the direct-object the way *it* wants to. The room's `look` then directly invokes the `look_self` verb on the object to be `look`ed at (the `ball`). (Later, you learn how verbs can invoke other verbs like this.) The `ball`'s `look_self` verb then prints out the text stored in its own `description` property.

look is programmed this way for extra flexibility so that in some rooms, for example, things can look different than they usually do! Don't be too daunted by look—it's an example of a pretty complicated program.

Examining a Verb's Argument Specifiers

You can see a verb's argument specifiers by using the @display (or just @d, for short) command.

```
@display obj:verb name
```

The @display command is useful for getting a variety of information about objects. The preceding form will get you information about a particular verb on a particular object (the asterisks will be explained in "Advanced Verb Syntax").

```
@d $thing:drop
    #5:"d*rop th*row"Wizard (#2)    rxd    this none none
------------------------------finished----------------------------
```

Huh? I Don't Understand That

There's one more wrinkle in how the server chooses which verb to invoke. Suppose that the server can't find any object with an appropriate verb (name and argument specifiers) on the player, nor in the room, nor the sentence's direct or indirect objects.

In this case, the server checks to see if the room has a verb named huh, and if it does, it invokes that verb! The huh verb is intended to be a last-ditch attempt to make sense of the sentence. If there is no huh verb, the server prints out:

```
I don't understand that.
```

The room's huh verb, like any other verb, could be programmed to do anything. In practice, it usually is programmed to understand a few kinds of slightly more complicated sentences than can the server. If huh can't figure out what the player meant, it prints out the same message that the server would have, as in the following:

```
I don't understand that.
```

huh is largely invisible to the both the player and the programmer, and you don't really need to worry about it. It's mentioned here to give you a more complete picture of how the server understands sentences. (If you come across a sentence that the server understands, but you're not sure *why* it worked, it might have been due to the huh verb!)

Note Objects

A note is an object that has writing on it, such as a slip of paper, or a posted sign. Notes are in the object class $note, the *generic* note.

```
@create $note named No Trespassing Sign,sign
```

```
You now have No Trespassing Sign (aka sign) with object
number #2171 and parent generic note (#9)
```

You can refer to your note by its long name, `No Trespassing Sign`, or by the short name you gave it, `sign`.

Your note will need some description…

```
@describe sign as "Stiff cardboard with a red and black warning.
```

```
Description set.
```

Finally, you presumably will want to write something on your note…

```
write "Violators Will Be Prosecuted" on sign
```

```
Line added to note.
```

You can use the `write` verb to write additional lines on the note, if you want.

Set down the note, and leave it in the room you're in. (Imagine that you are in a room called `Sure Would Forest`.)

```
drop sign
Dropped.
look
Sure Would Forest
Tall trees create a quiet and dark place of solitude. Sunlight
glimmers from above, and leafy shadows dance around you.
You see No Trespassing Sign here.
look sign
Stiff cardboard with a red and black warning.
There appears to be some writing on the note...
read sign
There appears to be some writing on the note...

Violators Will Be Prosecuted

(You finish reading.)
```

If you decide that you don't like the message on the sign, you can erase it, all at once, with the `erase` verb.

```
erase sign
```

```
Note erased.
```

Only the owner of a note can `write` on it, or `erase` it.

The `$note` object defines those verbs for us, and your `No Trespassing Sign` inherited them because it's a child of `$note` (#9). Their argument specifiers are:

```
read     this none none
erase    this none none
write     any on top of/on/onto/upon this
```

This means that the `read` and `erase` verbs work in simple sentences where the direct-object is your `No Trespassing` object. The `write` verb requires a preposition (introduce with any of the phrases `on top of`, `on`, `onto`, or `upon`) with an indirect-object that is your sign. Any words (`any`) will be accepted in the place of the direct-object.

A *$letter* is a Kind of *$note*

If you hand someone a `$note` object (that is, a child that you've created from `$note`), they can walk away with it and read it whenever they want. But when they are bored with it, what should they do with it? Because they are not the note's owner, they cannot recycle it; only you can do that. Perhaps they could just discard it by dropping it somewhere, but that leaves a mess in some (virtual) room. You could just wait for a couple of days and try to remember to recycle it for them, but there's no particular way for you to know whether they've read it yet.

Plain children of `$note` seem more suited for use as posters or signs, like your `No Trespassing` example. For a note intended for a particular player, you want some slightly different behavior.

The *generic* `letter`, `$letter`, is a child of `$note`. It inherits the `read`, `write`, and other verbs that `$note` has. If you create a child of `$letter`, your letter will in turn inherit those verbs from its parent. But `$letter` also has a verb that `$note` does not have: `burn`. Naturally, your child object will inherit `burn`, along with `read` and `write` and `erase`.

```
@create $letter named love letter

You now have love letter with object #2172 and parent
generic letter (#54).

write "My darling dearest," on love letter

Line added to note.

write "Please don't burn me again, but you can burn this
note as soon as you're done reading it." on love letter

Line added to note.
```

The `burn` verb on `$letter` allows anyone who can read the letter to also be able to `burn` it by typing

```
burn love letter

Love letter burns with a smokeless flame and leaves no ash.
```

The `burn` verb is programmed to recycle the object in a fashion similar to the `@recycle` command.

By the way, anyone could have read that love letter, and anyone could even have burned it! If you would like only a specific person or persons to be able to `read` and `burn` your `$note` (or `$letter`) objects, you can use the `encrypt` verb (defined on `$note`) to make such restrictions.

Help

The LambdaCore database, from which most MOOs are built, includes extensive help texts for users and programmers. You can read these simply by typing **`help`** and the name of the topic on which you would like help. For example, **`help notes`** would display the help text for `$note` objects.

`help`	Gives a list of basic help topics.
`help index`	Gives a list of major indices for help topics.

The Editor

The text of notes, textual property values, and verbs (programs), can be input and modified with the MOO's own simple text editor.

You begin "editing" an object by issuing the appropriate editing verb, such as `@notedit`. You are teleported into a special room that understands some verbs that are for text editing. Inside an editor, whether you are editing a note, a property, or verb, the editing verbs all work pretty much the same. You edit just one thing at a time inside the editor room, and when you're done, you leave the editor room.

To edit the writing on a note (or letter, or any other object descended from `$note`), use `@notedit`. You also can edit a textual property (for example, a description) using the `@notedit` command. When you write or edit a verb, you enter lines of program statements. To edit a verb, you use `@edit`, as in the following:

```
@notedit note obj

@notedit obj.property name

@edit obj:verb name
```

The editor is line-oriented. You enter lines of text, which are numbered. If you need to refer to a line (for example, to change or delete it), you simply refer to it by line number.

The editing rooms are programmed to make it seem like you're the only one inside, all alone. If you `look`, you will get information about help topics rather than a description of some physical place. Also, there's no talking or emoting among players in an editor room. Instead, the `say` verb in these rooms is programmed to take your words and add them to the text being edited.

```
@notedit sign

 Note Editor

 Do a 'look' command to get a list of the commands,
 or 'help' for assistance.

 Now editing "No Trespassing"(#2171).
```

To see what text is already on the note, use the `list` command:

```
list

__1_ Violators Will Be Prosecuted
^^^^
```

If there are many lines of text, `list` will only show you a few of them, unless you specifically ask for a certain range of lines. A range of line numbers could be

`1`	Just line 1
`4-17`	Lines 4 through 17.
`1-$`	Line 1, through the end (`$`). In other words, all the lines of text.

Adding Text

As you edit, the room keeps track of which line you are on. In the `list` command (mentioned previously), you can see that you are on line 1, indicated by the `^^^^` pointing at the line number. (So far, there's only one line of text on your sign, so of course you're still on line 1).

To add a new line of text right after the line that you're on now, just `say` the line that you want added.

```
"This means you.

Line 2 added.

list

  1: Violators Will Be Prosecuted
__2_ This means you.
^^^^
```

The line you're on sometimes is referred to as the *insertion point*, because it's where new text will be added. The insertion point *does not* move when you do a `list` command. Some other commands (such as `delete`) move the insertion point, but those commands tell you what action they have taken and which line you now are on, so you won't become confused.

You can move the insertion point around if you want to place new text somewhere other than the end. Use the `insert` command, and `say` which line comes after the place that you want your new line to go. For example, to insert something before the first line, type

```
insert 1

____
^^1^ Violators Will Be Prosecuted

"POSTED

Line 1 added.

list 1-$
__1_ POSTED
```

```
^^2^ Violators Will Be Prosecuted
  3: This means you.
```

The insert command doesn't actually add any text; the say (") command does that. The insert command just moves the insertion point to indicate where any new text should go.

The insertion point now is between line 1 and line 2. (Notice that all the lines are renumbered because of your insertion.)

If you want to add something to the end of a line (without making a brand new line), you emote. Move the insertion point back to the very end of your text, and add something to the last line.

```
insert $

__3_ This means you.
^^^^

: We're not kidding!

Appended to line 3.

list 1-$

  1: POSTED
  2: Violators Will Be Prosecuted
__3_ This means you. We're not kidding!
^^^^
```

Deleting and Changing Text

The simplest way of making changes, albeit the most cumbersome, is to retype the entire line that is in error. To do this, first delete the line you don't like, and then just type in a replacement line.

```
list 2

2: Violators Will Be Prosecuted

delete 2

---Line deleted.  Insertion point is before line 2.

"Violators Really Will Be Prosecuted
```

When you delete a line, the insertion point moves so that you're ready to add new text to replace what now is gone. (Also, all the lines are renumbered, but don't let that fake you out. When you add your new line back in, the lines are renumbered *again*! It all comes out in the wash.)

Substituting Text

You can use the subst command to make small changes within a single line or a range of lines, such as substituting one word for another. You specify the old text and the new text, delimiting them with a slash (/). Note that there's no space between the word subst and the first slash.

```
subst/old/new

subst/old/new/range of lines

subst/old/new/g

subst/old/new/grange of lines
```

If you don't say which line to change, subst will only make changes on the line that you're now on (the insertion point, to which ^^^^ points). To make changes across a range of lines, add an extra slash and specify the range you want (such as 1-$). When it's all finished, subst does an automatic list of the lines that changed.

subst is case-sensitive, which means that you have to type the letters of the old text in upper- or lowercase, just as it appears, or it won't be replaced. If you want subst to ignore capitalization, you can put the letter c after the ending slash to indicate that case doesn't matter.

Finally, remember that subst only makes one substitution on each line. You can put the letter g after the ending slash to indicate that you want global substitutions (more than one change per line). You also can combine the c and g modifiers.

Here's how you might add a little more text to your sign, and then finish off edits with a substitution. Replace in every line in the text all letter os with asterisks (*), regardless of whether they are upper- or lowercase, no matter how many there are on a line.

```
insert 2

__1_ POSTED
^^2^ Violators Really Will Be Prosecuted

"Keep Out

Line 2 added.

list 1-$

  1: POSTED
  2: Keep Out
  3: Violators Really Will Be Prosecuted
__4_ This means you. We're not kidding!
^^^^

subst/o/*/cg1-$

  1: P*STED
  2: Keep *ut
  3: Vi*lat*rs Really Will Be Pr*secuted
__4_ This means y*u. We're n*t kidding!
^^^^
```

Now the words in your sign appear as though perhaps it's been violently altered by someone who has good marksmanship skills and poor judgment!

save

When you finishing editing with `@notedit`, you have to `save` before leaving the editor, or your changes will not take effect.

```
save
```

A neat trick you can do in the note editor (when editing notes or properties) is to take the text from one object, and install it on a different object. This enables you to copy text, perhaps incorporating some changes, and then put it on a different object. To do this, simply `@notedit` the note or property that has the text you want to copy elsewhere. Make the changes you want to the text in the editor. When you're done, rather than doing a regular `save`, you can specify a different place to which to `save` your text.

```
save note obj
```

```
save obj.property name
```

If you save to a different object in this way, the editor knows that you've "changed your mind" about which note or property you want to edit. The plain old `save` command now will save text onto this different object that you've chosen. You can switch around to different objects with the `save` command as many times as you want.

Saving Changes to Verbs

The verb editor does not use the `save` command. Instead, use `compile`, which checks over the program you've entered, and either saves the verb, or else complains that there are mistakes.

Leaving the Editor

When you finish editing, use the `done` command to exit the editor room. Your changes will not take effect unless you do `save` (for notes and properties) or `compile` (for verbs) before leaving.

If you exit the room without first saving your changes, the editor remembers what you were doing and keeps your work around for you. If you come back to that editing room in the near future, your work will still be there, and you can pick up where you left off. (This is useful if you must leave the MOO to finish cooking dinner, or if you are disconnected accidentally.)

If you are inside the editor and decide that it was all a big mistake, you can use the verb `abort` to throw away your changes and leave the room. (Any `save` commands you already did already will have taken effect, but unsaved changes will be lost.)

Client-Based Editing

Some clients provide better editing capability than the MOO's editing rooms. If your client provides a better way to edit, you don't have to use the commands in the MOO editor to make changes. Because there are many different kinds of computers and many different clients, clients are beyond the scope of this book. However, here are some hints.

Cut and Paste

If your client supports `Cut` and `Paste`, and you have some editing program on your computer that you know and love, you may be able to use it in conjunction with the MOO's editor. This is somewhat primitive, but you might like it better than using only the editor commands mentioned previously.

You could capture the desired text (with `list` in the editor, or maybe even from a `look` outside the editor), and then paste it into the editor on your computer. Now make your changes there. Don't forget to strip out any line numbers and other junk that isn't actually part of the text!

To get ready to receive the text, go back to the MOO and enter the appropriate editing room (such as `@notedit`) on the object. Use the command **`delete 1-$`** to delete every line of text.

Now, to enter many lines of text all at once, use the `enter` verb. It accepts lines of text (without any `say` or `emote` verb) one after the other, until you enter a line that has nothing on it except a period (`.`). Do `enter`, and then paste all the replacement text from your editor onto the MOO. When it's there, `say` a line with just a period on it, and you're done.

TIP

If you get stuck inside an `enter`, type in a line consisting of the word `@abort`, and that should get you out.

Other Editing Commands

There are several other editor verbs for joining lines together, searching for text, and so on. For more information, use the `help` command while inside the editor.

Customizing with Messages

The main way that most objects respond to most of their verbs is to print a message to one or more players. For example, when you pick up any `$thing`, the `get` verb on the object prints out several messages. One message goes to you, and says something like, `You pick up ball`. Another message goes to everyone else in the room, `Reader picks up ball`.

Rather than putting the message right into the verb, where it could not be changed without reprogramming, it's more fun when players can change the messages around. To this end, the messages are remembered in properties on the object, and are inherited by the child object that you create. You can customize your (child) object by changing the contents of the message properties.

To find out all the messages that can be easily customized on an object, and to see what they are set to look like at the moment, use the `@messages` command. For example, there are several messages that are customizable on your own player object:

```
@messages me
```

```
@more me is "*** More ***  %n lines left.  Do @more [rest¦flush] for more."
@page_absent me is "%N is not currently logged in."
@page_origin me is "You sense that %n is looking for you in %l."
@page_echo me is "Your message has been sent."
```

You change the messages using the commands shown by `@messages`.

```
@page_absent me is "%N is off in the real world somewhere."
```

```
You set the "page_absent" message of Reader (#1007).
```

The `%N` in the message is a placeholder for your name, and will be processed by the verb when it's time to print the message.

Later on, you see how *pronoun substitution* works, and also how to program the messages for your own new types of objects.

Digging Rooms

Rooms are the objects on the MOO that represent the places in which you can wander around. Rooms are all descended from `$room`, the *`generic`* `room`.

You can create rooms the same way as other objects, with `@create`, but it's more common to use the special verb `@dig`. The difference is that `@dig` knows how to hook up entrances and exits for you (which would be tedious to do by hand after an `@create`).

A basic room has properties that `say` things such as who are the room's residents, what messages will be printed if someone is forcibly expelled from the room by the owner, and whether the room is dark. There are verbs on `$room` for: `say` and `emote`, that control how player speech and actions appear; `look`; and verbs that manage the `exits`.

The `@dig` command creates a child of `$room`. After you have created the basic room, you can use `@chparent` to change parents to some generic descendant of `$room`. Fancier rooms might have security, to enable the owner to control which players may enter, and when. Some rooms understand about `furniture` objects on which players can sit or lie down. Some rooms represent outdoor scenes, and may even be programmed to have `weather`.

```
@dig new-room-name
```

This `digs` a new room that is not connected to anyplace else.

```
@dig "Airport West Ramp"
Airport West Ramp (#172) created.
```

You should write down the objnum that `@dig` tells you because that is how you will have to refer to the room. (Because the room is not hooked up, you can only get there by *teleporting* with the `@move` command!)

The `new-room-name` should be either a single word with hyphens (as in the preceding example), or you need to put the words in quotes.

```
@dig exit-spec to new-room-name
```

This `digs` a room (as in the preceding example), but also creates an exit from the room you're in to the new room. The `exit-spec` indicates which exits should be created and hooked up.

If you simply use a direction name, like `west`, `@dig` will create an exit named `west` from the room you're in to the new room. People are accustomed to abbreviating `west` to just `w`, so you should probably arrange for that to work by using the `exit-spec west,w`.

If you want to create exits from your new room to the room you're in, use the `exit-spec west,w¦east,e`.

In addition to `north`, `south`, `east`, `west`, `northeast`, `northwest`, `southeast`, and `southwest`, you can use almost any word as the name of an exit direction. (The room's `huh` verb will make this work.) Some other common ones are `out`, `leave`, `up`, and `down`. People also sometimes give descriptive aliases to a direction; the exit-spec `up,stairs,leave,out` enables the player to type any of those words to go in the same direction.

`@dig` tells you the objnum of the exits it created, but you don't need to write those down or remember them. You usually don't need them, but you could always find them again with the `@exits` command.

```
@dig exit-spec to room
```

Once you've created a room, this form of the `@dig` allows you to add more exits and entrances to it.

Exits

Every room has a property called `exits` that contains a list of all the exits (that is, it contains a list of objnums of the relevant exit objects). The `exit` objects, in turn, have properties that remember where they connect to. This information forms the geography of the MOO's virtual world.

When you type `west`, for example, the server does its usual thing and finds a verb named `west` on the room. The room actually has verbs named `east`, `west`, `north`, `south`, and so on, and all those verbs do the same thing. The verbs look inside the room's `exits` property for an exit that has the same name, and then they in turn `invoke` a verb (somewhat confusingly named `invoke`) on the exit. The `exit` object has a property named `dest` (destination), which remembers the objnum of a room. The invoke verb on the exit does the work of transporting the player into that room, and also prints out a message.

You can get a list of the exits in a room with the @exits command:

```
@exits

south (#102) leads to Yellow Room (#101) via {south, s}.
up (#90) leads to Second Floor Landing (#129) via {up, u}.
east (#153) leads to Sandy Castle (#112) via {east, e, out}.
```

To find out about messages that you can set on those exits, use @messages:

```
@messages up

@oleave up is "%N ascends the spiral staircase."
@leave up is "You begin climbing the spiral staircase."
@nogo up isn't set.
@onogo up isn't set.
@arrive up isn't set.
@oarrive up isn't set.
```

Exits also have description properties, and you should @describe them, so that if someone looks at them (looks in that direction), they see a door, a cliff, a water slide, and so on. **@describe up** might evoke A beautiful spiral staircase leads up.

(Note that although the exit named up is stored in the exits property of the room, so it in some sense is a part of the room, but that's not the same as being in the contents of the room. How were you able to refer to the up exit, since it's neither on your person nor inside the room? How did the server figure out which object up is? Actually, the server never did figure it out. Rather, this is an example of the room's huh verb getting to look at your sentence. The room's huh is programmed to know about the room's exits. Note that this entire process was invisible! You don't need to worry about it.)

Have a Ball

Return to the ball you created earlier.

```
look ball

A bouncy rubber ball.

@parents ball

ball(#2361)  generic thing(#5)  Root Class(#1)
```

By virtue of being a $thing, the ball has inherited the verbs get and drop (also known as throw), so you can pick it up and put it down, but not much else. It would be a lot more fun if you could do more things with it.

```
@verb object:verb name   argument specifiers
```

You must use @verb only the first time when you go to define a new verb for an object.

```
@verb ball:bounce this none none

Verb added (0).
```

This creates a `bounce` verb on the `ball` that will match the sentence, `bounce ball`; for this verb, the direct-object must be this ball.

Of course, simply telling the MOO that the `ball` will have a `bounce` verb isn't quite enough. Trying to `bounce` the `ball` at this stage of the game will yield the following confusion:

```
bounce ball

I don't understand that.
Try this instead:  bounce ball
```

In order for it to do anything useful, you need to program the `bounce` verb on the ball. Generally speaking, a program is nothing more than a sequence of the exact steps that should be taken under a given circumstance. Before you can program the bouncing ball, you need to decide what it *means* to `bounce`—what do you want to have happen?

When someone bounces the ball, you would like to see a message to that effect, and you would like other players in the same room to see it, too.

To write the program, you go into the verb editor, type in your program, and `compile` it.

```
@edit ball:bounce

Verb Editor

Do a 'look' to get a list of commands, or 'help' for assistance.

Now editing #2361:bounce (this none none).

enter

[Type lines of input; use '.' to end or '@abort' to abort the command.]

#1007:tell("You bounce the ball");
.

Line 1 added.

compile

#2361:bounce successfully compiled.

done
```

You can use the `@list` command to see the program for a verb.

```
@list ball:bounce

#2361:bounce this none none
 1:   #1007:tell("You bounce the ball");

bounce ball

You bounce the ball.
```

Congratulations! That's all there is to it—you now have your own MOO creation—a ball that bounces!

However, there's quite a bit of room for improvement.

@program

There's another way to type a verb's program into the MOO, without using the verb editor (`@edit`). The `@program` command is a shortcut, but it provides no way to edit. If you're not going to make any mistakes (perhaps because you're pasting the program from your client, rather than typing it), `@program` is more to the point. When you finish inputting, `@program` automatically does the same thing that `compile` and `done` does in the verb editor.

```
@program ball:bounce

Now programming ball:bounce(0)
[Type lines of input; use '.' to end or '@abort' to abort the command.]

#1007:tell("You bounce the ball");

.

0 errors.
Verb programmed.
```

Verb Examples

Further examples show just the program (code) itself, without bothering to show the excursions into the verb editor; nor is the `@verb` command shown. Instead, you will just see the code as `@list` shows it; from this, you can easily figure out how to type the program. You've been there, done that.

Simple Statements

A program in the MOO language is a series of *statements*. Most statements are one line long, and end with a semi-colon (;).

So far, the example program shown previously is very simple, consisting of just one statement:

```
#1007:tell("You bounce the ball");
```

This typical statement looks a little bit like English, and you can easily guess what it's all about. Your own `player` object (for `Reader`) is `#1007`, and you somehow are to be told that `You bounce the ball`.

Calling All Verbs

Much of the work on the MOO is accomplished by verbs calling on other verbs to do part of the desired work.

Verbs such as `bounce` are intended to be called by the server when it translates a sentence typed by a player.

Other verbs, such as `tell`, are subroutines; they're intended to be invoked directly by other verbs, rather than by the server.

Subroutines

Programs are complicated. Even something as simple as printing out a message requires many steps. If every verb had to do all the work by itself, it would be both tedious and tricky to write, and all the verbs would be long and contain duplicated code.

Programs sometimes are called *routines* because they spell out exactly what rigamarole is to happen under a given circumstance. The idea of a *subprogram* or *subroutine* is that common operations, such as printing out a message, can be written just once. When a program wants to perform one of the common operations, it refers to the subroutine instead. A subroutine is thus a kind of "helper" or "utility" program.

A verb can call another verb with a statement like the following:

```
object:verb name(argument 1,argument 2,argument 3 ...)
```

The `object` references the MOO object in which the verb named `verb name` is defined. The colon separates the object from the verb name. The arguments (information to be passed on to the verb) go inside the parenthesis, separated by commas, in the order the verb expects them.

tell

Every MOO object has a `tell` verb, which does the job of `tell`ing the object which words are being spoken. On `players`, `tell` is programmed to print out a message that you can read. On most other types of objects, `tell` doesn't do anything. But `tell` also might make sense, for example, if the object was some type of a (virtual) tape recorder.

The `tell` verb takes one argument: the message that is being told.

```
#1007:tell("You bounce the ball");
```

This statement means to call the verb `tell`, defined on `#1007`, with the argument `"You bounce the ball"`. Of course, `#1007` is the objnum of your player, and its `tell` verb prints out the message.

Using Variables

One glaring problem with the `bounce` verb is that it has `#1007` hard coded into it. If someone other than yourself tries to `bounce the ball`, they won't get any message, because the program says to `tell` the message only to `#1007`. This would be doubly confusing, because *you* would see the message `"You bounce the ball"`, even though it was someone else bouncing it! The problem is that the person who should see the message varies, depending on who typed **`bounce ball`**.

You can make this work by using a variable. A *variable* is a name that refers to some temporary information. A variable only has information during the execution of a verb, and effectively ceases to exist once the verb concludes.

Properties on objects keep information indefinitely, but *variables* in verbs only have information while the verb is executing.

Programmers can make up any new variables they want in their verbs, and there also are several special "built-in" variables to which a verb can refer. Perhaps the most important built-in variable is `player`, which has the objnum of the player invoking the verb.

If you change the program from

```
#1007:tell("You bounce the ball");
```

to

```
player:tell("You bounce the ball");
```

then the player who invokes `bounce` will be the one who sees the message.

Announcements

Another problem with the `bounce` verb is that only the player who `bounces the ball` will see a message. It would make more sense if the other people in the same room also could see that someone was `bouncing a ball`.

Add the following statement to your program:

```
player.location:announce(player.name," bounces the ",this.name);
```

This statement, which is another verb call, illustrates several basic things.

```
@list ball:bounce
#2361:bounce   this none none
  1:   player:tell("You bounce the ball");
  2:   player.location:announce(player.name," bounces the ",this.name,".");
```

The way to write a reference to a property, to get at a property's *value*, is to simply write

```
object.property name
```

In this case, you are referring to the `location` property of the object referred to by the player. The `location` property of an object remembers which object is inside; the player's `location` property is a room object. In other words,

```
player.location
```

refers to the room the player is in.

The `announce` verb defined by the room is programmed to go around to every object in the room and `tell` it something. `announce` takes any number of strings and appends them together, and calls `tell` with the resulting message on every object in the room, except that it skips the player who invoked the verb.

A *string* is just a collection of characters (letters, numbers, spaces, and so on). A sentence, for example, is a string. You write strings inside quote marks: `"like this"`.

Another built-in variable is `this`, which refers to the object that defines the verb. So, when the `bounce` verb defined by your ball (`#2361`) is executing, `this` refers to `#2361`. `this` is how an object references itself.

The `name` property of an object remembers the name given to it (usually by the `@create` or `@rename` command).

So `this` line of the verb causes a message like

```
Reader bounces the ball
```

to be displayed to everyone else in the room.

While you're at it, you might as well take out the word `ball` from the string it `tells` the player, and replace it with a reference to the ball's name, as you did for `announce`. The program now looks like the following:

```
#2361:bounce   this none none
 1:     player:tell("You bounce the ",this.name,".");
 2:     player.location:announce(player.name," bounces the ",this.name,".");
```

This works the same way as before, but if you change the name of your object to something more interesting than `ball`, the messages then come out with the new name.

```
bounce ball

You bounce the ball.

@rename ball to beach ball,ball

Name of #2361 changed to "beach ball", with aliases {"beach ball", "ball"}.

bounce ball

You bounce the beach ball.

@describe ball as "A multi-colored beach ball.

Description set.
```

More Built-In Variables

Several built-in variables exist for all verbs to use. Most of them have to do with the command sentence the player typed, and which objects the server found that matched. You've already seen how `this` and `player` are commonly used:

`this`	The object on which this verb was found (Object)
`player`	The player who typed the command (Object)

There are also the following:

`dobjstr`	The direct object string (String)
`dobj`	The direct object value found by the server (Object)

`prepstr`	The prepositional phrase (String)
`iobjstr`	The indirect object string (String)
`iobj`	The indirect object value found by the server (Object)

As you can see, there are two variables for each part of speech. One is the string of words, and the other is the corresponding object (that is, an `objnum`) that the server located.

Sometimes the server will not be able to find any object that matches the words for a part of speech. When that happens, a special object called `$nothing` is used as a placeholder in the variable. Verbs can test to see if, `dobj`, for example, is `$nothing`, meaning that no object was found corresponding to `dobjstr`.

As far as argument specifiers are concerned, `$nothing` is something. That is, an argument specifier of `any` will match `$nothing`, just like it would match a valid object.

Other Built-In Variables

Here are some more variables that are needed by some more complicated programs, beyond the scope of this book:

`verb`	The first word of the command sentence (String)
`argstr`	Everything after the first word (String)
`args`	A list of strings (The words in `argstr`)
`caller`	The same as `player` (Object)

Subroutine Argument Specifiers

Some verbs are not intended to be invoked when players type sentences. These verbs are only for use as subroutines, and are only supposed to be called by other verbs.

The following is the argument specifier for this kind of subroutine verb:

```
this none this
```

You can see that this argument specifier could never match any sentence: it has a direct object, and also an indirect object, but not a prepositional phrase. This nonsense combination was chosen to represent a subroutine verb that cannot be invoked by a sentence.

Getting Rid of Verbs

If you mistype an `@verb` command, or if you change your mind about wanting to define a particular verb, you can erase the verb from the object with the `@rmverb` command. Not only does the verb name and argument specifiers go away from the object, but also any code that was programmed for the verb is forgotten.

```
@rmverb object:verb name
```

```
@rmverb object:verb name argument specifiers
```

This command also is useful if you accidentally use `@verb` with the same verb name more than one time on the same object. `@rmverb` erases the most recent verb that matches the name (and optionally, that also matches the argument specifiers.)

if Statements

The way you've programmed your beach ball, anyone on the MOO can play with it. That's nice, but the problem is, they don't have to have the `ball` in their possession in order to `bounce` it. In fact, they don't even have to be in the same room as the `ball`! They could be in some other room, but if they do **`bounce #2361`**, they will get the message that they are bouncing the ball, and people in the room in which the ball is will see the message about someone bouncing the ball. Clearly, this doesn't make sense!

You can fix this by programming `bounce` to check to see whether the player is already holding the ball. If they aren't, they should get a complaining message to that effect.

Testing to see if some condition is `true` or not, and taking alternative actions is a fundamental programming technique that's used all over the place. Rewrite the ball's `bounce` verb to look like the following:

```
#2361:bounce   this none none
   if (this.location == player)
     player:tell("You bounce the ",this.name,".");
     player.location:announce(player.name," bounces the ",this.name,".");
   else
      player:tell("You can't bounce a ", this.name,
     " that you haven't got.");
   endif
```

This verb illustrates the classic `if-then-else` type of statement that is common in almost all programming languages.

```
if (expression) then stuff endif
```

```
if (expression) then stuff else other stuff endif
```

The `expression` can be any value, referencing a variable, a property, or any type of computation that you can express in a simple MOO statement. It always goes inside the parenthesis, and determines how the subsequent statements in the verb flow. If the `expression` is `true`, the following statements will be executed. If the `expression` is `false`, the verb will skip down to the `else` (if there is one), and do the alternate statements instead. The `if` statement is concluded with an `endif`.

The following expression

```
this.location == player
```

compares the `location` property of the `ball` to the `player`'s object. If they are the same (if the location of the `ball` is the `player`), that means the `player` is holding the `ball`. In that case, this expression evaluates to `true`. `==` compares two values and returns a `true` value if they are equal.

Notice that the `if`, `else`, and `endif` words do not end with semicolons. Simple statements in the body of the `if-then` statements still end in semi-colons, as usual, but the special words that delineate them do not.

elseif

You perform several tests in succession in an `if` statement by using the `elseif` clause.

```
if (expression)
   then stuff
elseif (expression)
   some other stuff
else
   last-ditch other stuff
endif
```

You can include as many `elseifs` as you want. However, there can only be one `else` statement, and it has to come last, or not at all.

Properties

You now have a nice bouncy beach ball that anyone can play with, as long as they are holding it, but you can make it more interesting by giving it more properties, and teaching its `bounce` verb to use them.

A more interesting kind of ball is one that could be inflated and deflated. Begin by adding a property that remembers whether or not the ball is inflated.

```
@property ball.inflated 0
```

```
Property added with value 0.
```

This adds a property named `inflated` to your ball, and gives it a value (for now) of `0`. On computer systems, `true`/`false` values typically are represented by the numbers `0` (for `false`, `no`, or `off`), or `1` (`true`, `yes`, or `on`). In particular, the MOO language considers `0` to be a `false` value.

You have one `if` statement that tests to see whether the player is holding the ball. Rewrite the `bounce verb`, putting another `if` statement inside that one that tests whether the ball is inflated:

```
#2361:bounce   this none none
 1:   if (this.location == player)
 2:     if (this.inflated)
 3:       player:tell("You bounce the ",this.name,".");
 4:       player.location:announce(player.name," bounces the ",this.name,".");
 5:     else
 6:       player:tell("You have to inflate a ",this.name,
                      " before it will bounce!");
```

```
    endif
  else
    player:tell("You can't bounce a ", this.name,
                " that you haven't got.");
  endif
```

Because you set the `inflated` property to `false` (`0`), trying to bounce the ball no longer works:

```
bounce ball
You have to inflate it before it will bounce!
```

You can change an object's property to have a new value with the `@set` command:

```
@set object.property to value
```

Look at the following example:

```
@set ball.inflated to 1
Property #2361.inflated set to 1
bounce ball
You bounce the ball.
```

Whitespace

When typing in your MOO programs, spaces and line breaks (the space bar and the Enter key, respectively) are interchangeable. You can use whichever one suits you, and you can type as many statements on one input line as you want. Regardless of how you enter your program, the server makes the appropriate type of whitespace to break up the individual lines, so that it always comes out looking good.

The exception to this is strings. You can't put a line break (you are not allowed to press Enter) in the middle of a string. If a string is too long to fit on one line, it just wraps onto the next line.

Verbs for Setting Properties

Using the `@set` command destroys the illusion of virtual reality; there's nothing like `@set` in real life. In real life, you would inflate the ball by blowing it up with your breath or something. Moreover, `@set` only works on objects that you own. If someone else tries to `@set` your ball's `inflated` property, they will get a `M Permission Denied` error message.

What you want is to have a verb that anyone could use to inflate and deflate the ball. Call these new verbs `inflate` and `deflate`.

```
#2361:inflate   this none none
  if (this.location == player)
    if (this.inflated)
      player:tell("It's already inflated.");
    else
```

```
 5:       player:tell("You pucker up and inflate the ", this.name,".");
 6:       player.location:announce(player.name,
                                   " puckers up and inflates ",this.name);
 7:       this.inflated = 1;
 8:     endif
 9:   else
 10:    player:tell("You can't inflate a ", this.name,
                    " without picking it up, first.");
 11:  endif

#2361:deflate   this none none
 1:   if (this.location == player)
 2:     if (this.inflated)
 3:       player:tell("It's already deflated.");
 4:     else
 5:       player:tell("You deflate the ", this.name,".");
 6:       player.location:announce(player.name," lets the air out of the ",
                                  this.name);
 7:       this.inflated = 0;
 8:     endif
 9:   else
 10:    player:tell("You can't deflate a ", this.name,
                    " without picking it up, first.");
 11:  endif
```

These verbs match sentences, such as `inflate ball` and `deflate ball`. They have a similar layout to the `bounce` verb. The outermost `if` statement checks to see whether the player is holding the ball, while the `if` statement nested inside the first `if` statement checks to see whether the ball has already been inflated.

The expression on line 7 of `inflate`

```
7:      this.inflated = 1;
```

assigns the value `1` (which is considered to mean `true`) to the `ball`'s `inflated` property. The corresponding line in `deflate` sets the property back to `0` (`false`) to indicate the ball is not deflated.

Like the `ball`'s `bounce` verb, any player (not just the owner) can now `inflate` or `deflate` the ball, but only if he or she is holding it.

Message Properties and Pronouns

The messages that the beach ball prints are *hard coded* into it, meaning that the verbs have to be rewritten if you want to change the messages. It would be better programming style to put the messages in properties, where they can easily be changed.

You can name your properties anything you want, but there is a convention for naming message properties. If you follow the standard convention, the `@messages` command and the message-setting commands will work. Also, other programmers looking at your object will more readily understand how your `ball` works.

The main thing is that name of each message property should end in `_msg`. For example, some message about saying hello should be stored in a property named `hello_msg`.

So far, your `ball` has messages for when you `inflate`, `deflate`, and `bounce` the ball. There's one version for the `player`, and one version for the other people in the room. Then there are messages for when those actions don't work (you can't `bounce` the `ball` because it's `deflated`, for example). Finally, there are messages for when you try to do something with the ball but you are not already holding it. The latter hard codes remain the same because they don't seem to make much sense changed around, but all other hard codes are turned into properties.

The `message` properties should be named in such a way that players can tell with which message the verb the message goes. Messages sent to other people in the room should begin with the letter `o` (for `others see…`). Messages about things that don't work include the word `fail`.

```
@property ball.bounce_msg "You bounce the %t".
@property ball.obounce_msg "%N bounces the %t".
@property ball.bounce_fail_msg
          "You have to inflate a %t before it will bounce!

@property ball.inflate_msg "You pucker up and inflate the %t."
@property ball.oinflate_msg "%N puckers up and inflate the %t."
@property ball.inflate_fail_msg "It's already inflated."

@property ball.deflate_msg "You deflate the %t."
@property ball.odeflate_msg "%N lets the air out of the %t."
@property ball.deflate_fail_msg "It's already deflated."
```

Pronoun Substitutions

To allow the most flexibility in the messages, we're using a facility called *pronoun substitution*, where the magic tokens `%t` and `%N` are replaced by the appropriate words.

There is an object on the MOO that can be referenced with the special notation `$string_utils`. This object has a variety of verbs on it that are intended for programmers to use as subroutines. This collection of *string utilities* includes the verb that does pronoun substitution.

You run your string through `$string_utils:pronoun_sub`, and it produces a new string for you with the pronoun tokens replaced with the desired nouns or pronouns. The substitution for `%N` is the capitalized name of `player` (same thing as `player.name`), and the substitution for `%t` is the name of the direct-object (same as `C this.name`).

Now you have to rewrite the verbs to use the preceding properties, changing hard-coded strings to `$string_utils:pronoun_sub(this.` *`message property`*`)`.

```
#2361:bounce   this none none
if (this.location == player)
  if (this.inflated)
    player:tell($string_utils:pronoun_sub(this.bounce_msg));
    player.location:announce(
         $string_utils:pronoun_sub(this.obounce_msg));
  else
    player:tell($string_utils:pronoun_sub(this.bounce_fail_msg));
    endif
  else
     player:tell("You can't bounce a ", this.name,
                 " that you haven't got.");
 endif

#2361:inflate   this none none
  if (this.location == player)
    if (this.inflated)
      player:tell($string_utils:pronoun_sub(this.inflate_fail_msg));
    else
      player:tell($string_utils:pronoun_sub(this.inflate_msg));
      player.location:announce(
         $string_utils:pronoun_sub(this.oinflate_msg));
      this.inflated = 1;
    endif
  else
    player:tell("You can't inflate a ", this.name,
                " without picking it up, first.");
  endif

#2361:deflate   this none none
  if (this.location == player)
    if (this.inflated)
      player:tell($string_utils:pronoun_sub(this.deflate_fail_msg));
    else
      player:tell($string_utils:pronoun_sub(this.deflate_msg));
      player.location:announce(
         $string_utils:pronoun_sub(this.odeflate_msg));
      this.inflated = 0;
    endif
  else
    player:tell("You can't deflate a ", this.name,
                " without picking it up, first.");
  endif
```

More About Properties

When a property is inherited, the child gets its very own property of the same name. The child's property can have a different value than the parent, but initially the property is `clear`.

A `clear` property has the same value as does the parent's property. When the parent's property changes value, so does the child's; a clear property reflects the parent's property.

When a child's property is altered to some new value, it's no longer `clear`. From that point on, it has its very own value, independent from the parent.

Sometimes it's desirable to make a property clear again, getting rid of its unique value, and having it just reflect the parent's value. This is done with the `@clearproperty` command.

`@clearproperty` *`object.property`*

Removing Properties

If you decide to change around the way an object works, and no longer need one of the properties you had added to it, you can remove the property from the object.

```
@rmprop object.property
```

Built-In Properties

Objects have a few special properties that can't, or shouldn't be, manipulated with the `@set` command. In most cases, there are special commands for setting these properties. Setting these by hand either won't work or might result in a mistake that would be very confusing for you.

The `name` and `aliases` properties of objects are best manipulated with the `@rename` command.

The `contents` property remembers what things are inside the object (whatever that means). The trick here is that if object `A` is in the `contents` list of object `B`, then object `A`'s `location` property should be object B; they should be consistent. Also, it doesn't make sense for some objects to be inside other objects, and strange things could happen. The `@move` command knows how to keep these things straight.

The `permissions` properties of an object indicate what rights, if any, other players have to mess with the object; use `@chmod` to control these.

The `owner` property indicates which player owns the object, is set with the `@chown` command. (This command is only available to wizards. Regular players need the help of a wizard to change ownership. On some MOOs, wizards have created an "Ownership Transfer Station," a special room programmed to provide automated assistance in this area.)

NOTE The `programmer` property of a player can only be set by a wizard.

Advanced Verb Syntax

A single verb definition can match more than one verb in a sentence, if you want. This can be used to make synonyms. The `drop` verb on `$thing`, for example, is the same verb as `throw`. Rather than define two verbs that do the same thing, there is only one verb. The trick is to put spaces in the verb name, which are taken to mean that any of the indicated words should invoke the verb.

The name of the familiar `drop` verb actually is

```
"d*rop th*row"
```

Either `drop` or `throw` would match the verb in a sentence. You can use either word as the verb name in a command.

The asterisks (`*`) indicate how the verb can be abbreviated in a sentence. You can type `th` for `throw`, for example. This will only work if there are not conflicting verb abbreviations.

Prepositions

Verbs that allow prepositional phrases have argument specifiers, such as

```
this "with" any
```

meaning that a sentence with a direct-object is referring to this object, and a prepositional phrase `with` is referring to anything. For example, if you defined a `puncture` verb on your `ball`, using the preceding argument specifiers, the sentence `puncture ball with needle` would match. Likewise, you could define a verb called `toss` with the following argument specifiers

```
toss     this at any
```

to match sentences such as `toss ball at Kent`.

Alternatively, an argument specifier such as

```
any "with" this
```

matches sentences in which the indirect-object was this object. If you had a `hammer` object, you might define a `hit` verb with this argument specifier, so that you could `hit ball with hammer`.

The following is a complete list of prepositions:

```
with/using
at/to
in front of
in/inside/into
on top of/on/onto/upon
out of/from inside/from
over
through
under/underneath/beneath
behind
beside
for/about
is
as
off/off of
```

Making Generic Objects

Generic objects are objects that are not intended to actually be used, but are for making children. Generic objects typically start out as a regular one-of-a-kind object like your `beach ball`, and then are re-programmed so that any children will be easy for their owners to customize. You've done this with the beach ball by making it possible to change the messages it gives when it's bounced. The last step in "genericifying" your ball is to make sure it has the right permissions.

Ownership and Permissions

Every object, verb, and property, is owned by someone (usually the person who created it). Owners have control over their own objects, can add or remove verbs and properties, reprogram verbs, and alter the values of their properties. Owners also can change the *permissions* of their objects, properties, and verbs, to enable other players to do certain things with them.

Every object has a bunch of permissions that dictate certain ways that it can be used by other players on the MOO. It's important to understand permissions so that you can maintain appropriate control over your objects.

You change an object's permissions with the `@chmod` command, as in the following:

```
@chmod object permissions
```

```
@chmod object.property permissions
```

```
@chmod object:verb permissions
```

The `permissions` are specified as the letter of the permission, such as `r` for `readable`, preceded by a plus sign (+) to turn the permission on, or a minus sign (-) to remove the permission.

You can check the `permissions` of an object with the `@display` (abbreviated `@d`) command.

`@display` *`object`*

The preceding line tells you the object number, object owner, object permissions, and a few other useful things about the object.

`@display` *`object`*`.`

The preceding line displays information about the objects properties, including the owner and permission of each, and a (possibly abbreviated form of) each one's value.

`@display` *`object`*`:`

The preceding line displays information about the object's verbs, including the owner, permission, and argument specifiers of each.

Object Permissions

Objects can be `readable` (`r`), meaning that anyone can see the verbs and properties the object has. Object also can be `writable` (`w`), meaning that anyone (not only the owner) can add verbs and properties. You never really want an object to be writable, for reasons that are explained shortly.

An object also can be `fertile` (`f`), meaning that players other than the owner can create children from it. (If an object is not `fertile`, then it cannot be the parent in a `@create` or a `@chparent` command.)

Most objects you intend to share should be readable, so that other players can see the parts of the object, and get some idea of what it does.

```
@chmod ball +r
```

The preceding line makes the ball readable.

To enable other people to make their own objects, using your object as the parent, you need to make yours `fertile`:

```
@chmod ball +f
```

The preceding line makes your `beach ball` a `fertile` object. Other players now can `@create` their own `beach balls`, using yours as the parent.

Property and Verb Ownership

Every property and verb on an object is owned by some player. When those verbs and properties are inherited by a child object, they still are owned by the original player.

The `drop` verb on `$thing`, for example, is owned by a special player named Wizard (#2). Your `beach ball` inherited that `drop` verb, but `drop` on your object is still owned by Wizard.

The verbs and properties that you have defined on your `ball`, such as `bounce`, are owned by you. If someone creates a child of your `ball`, that child will have a `bounce` verb owned by you, and a `drop` verb owned by Wizard.

Verb Permissions

Verbs can be `readable` (`r`), meaning that anyone can list their program, or `writable` (`w`), meaning that anyone can change their program.

When a verb on your object executes, it does so with your power of attorney. It's as though you were performing the actions of the verb, legally speaking. Anything that you have permission to do, your verb can do.

Writable Verbs Are Bad

It's extremely important not to have `writeable` (`w`) verbs. If some object had a `writeable` verb owned by you, anyone could `write` (reprogram) the verb to do whatever they wanted. Because you would still be the owner of the verb, their program would have all the permissions that you have!

For example, only you have permission to change your player object's description (with **`@describe me as...`**, for example,). This also means that any verb you own also has this same permission. If you had a `+w` verb, any player could reprogram it to change your description without your knowledge (or send MOO e-mail with your name on it, or recycle an object you own, or do any other thing that you could do). It could even change around the permissions of your other objects, making *more* of your verbs and properties `writeable`!)

Why Verbs Need Your Permission

Verbs need to have your capabilities in order to get at the properties and other objects that you also control.

Verbs on objects that you want other people to use, especially `fertile` objects, should be `readable`. This way, other programmers can see how the object works. This enables the other players to figure out how to improve the object (by adding their own verbs to their children) if they want, and also provides some confidence that your verbs won't do anything undesirable.

Other Verb Permissions

Two more permission bits apply to verbs. These are not really permissions having to do with ownership. Instead, these extra permissions specify how the verb behaves. (This affects the verb owner as much as anyone else.)

The `execute (x)` permission allows a verb to be called by another verb, and is needed for subroutines such as the `announce` verb on rooms. Normally, verbs do not have this permission.

The `debug (d)` permission controls what happens to a verb it blows up due to a programming error. Verbs usually have `debug` permission turned on. This means that if the verb blows up, it will print out a `backtrace`, which is useful information to be given to the programmer. If the `debug` permission is turned off, the verb will just silently blow up without giving the player any `backtrace`.

Property Permissions

Properties have the trickiest kind of permissions. A property can be `readable (r)`, meaning that anyone can read its value, and `writable (w)`, meaning that anyone can alter the value.

Writable properties are not used very often, but do not represent the same kind of security problem as do writable verbs. You probably don't want to make your object's properties writable.

A property that is not readable cannot be `read` by anyone except the property's owner (you, for example). However, verbs owned by the same player can get at those properties, just as the player could. In more complicated types of objects, it's important that players not be able to read properties directly. Instead, a verb would be provided for them to use the information.

Unreadable Properties

An object, for example, might have a property called `dates` that keeps track of secret romantic dates made between players. The `dates` would be unreadable (`-r`), so nobody (except the object's owner) could read it directly. The object would have a `rendevouz` verb that would tell you when and where your admirer wanted to meet you, and allow you to complete making a date with that player. Naturally, the `rendevouz` verb would be programmed to check which player you were, and only give you information about your own dates. Because `rendevouz` and the `dates` property are both owned by the same person, `rendevouz` can both `read` and `write` the secret information in the property. The information is kept secret, even though anyone can use `rendevouz`. Normally, however, properties are readable.

Inherited and Changeable Properties

When a child is created from an object, all of the parent's properties are inherited, too. This means that the child will have properties with the same name as all the properties that its parent has. The child's properties are separate from the parent's properties; they just have the same name.

In the virtual world, the child represents a different object than its parent. Even though two beach balls might look the same, and do the same kinds of things, they are certainly not the same object. The properties remember the object's location, description, and everything else about it. So each object's properties are unique to it.

Even though an object's properties are its own, however, those properties might not necessarily be owned by the object's owner. Each property is or is not `changeable`. The `changeable` (`c`) permission determines who controls the property.

If a property on a parent is not changeable, then the parent continues to control the property on its children. Each child inherits the property, but the child cannot control it: only the parent can alter the value of the property. The parent remains the owner of the inherited property. This allows the parent object's owner to retain some control over the children.

Suppose that you had a `generic doll` object with a property called `hair_color`. If that property was not changeable, when someone created their own doll, that player would not be able to change the doll's hair color. Instead, perhaps the parent would have a `dye` verb that alters the `hair_color` property, but would only allow certain colors. The child inherits the `dye` verb, which also is owned by the property. Any player can use `dye` to change the `hair color` of his or her doll, but he or she can't write their own verb to directly alter their doll's `hair_color`. The `dye` verb works because both `dye` and `hair_color` are owned by the parent, and they are designed to work together.

By contrast, if the `hair_color` property on the generic `doll` was changeable, then any child `dolls` would have direct control over their hair color. That player could, for example, use the `@set` command, or write their own verbs, to alter the `hair_color` property.

Changeable properties are owned by their object owner, not the parent's owner. For example, the `drop_succeeded_msg` property on your ball, was inherited from `$thing` (which is owned by a player usually named Wizard). But `drop_succeeded_msg` is a changeable property, so when it appears on your ball, it's owned by you, `Reader`.

Generally speaking, properties that are used as messages should be changeable.

The `@property` command, which creates new properties, normally sets the permissions to `rc` (readable and changeable) so that anyone can read the properties and so that children can change their own properties' values.

```
@d ball.bounce_msg
.bounce_msg                 Chris (#91)          r c    "You bounce the %t".
------------------------------finished---------------------------
```

Your `beach ball`'s properties are readable and changeable, as they should be. When a player makes a child of your `beach ball`, he or she will be able to change its description and its messages.

Nonchangeable Properties

Not all properties should be changeable: sometimes the parent needs to have control over them. A classic example from the early days of LambdaMOO involves a `radio`. The `radio` had a verb `tune` that `tuned` the `radio`, and a property `channel` that remembered the channel to which it was set. The `tune` verb would, of course, change what was in the `channel` property. The owner of the `radio` object did **`@chmod radio +rf`**, making it readable and fertile, so that others could make their own `radios` from the parent `radio`.

Mysteriously, the `tune` verb didn't work on any of the child `radio` objects. The problem was that the `channel` property was changeable (had `+c` permission); the owners of radio objects owned the `channel` properties on those objects. The `tune` verb, however, was owned by the original parent `radio`'s owner. `tune` could not change the `channel` in anyone else's `radio`—that property had a different owner on every individual `radio`.

To fix this, the owner of the parent `radio` made the `channel` property `unchangeable` by doing **`@chmod radio.channel -c`** on the parent `radio` object.

That fixed the problem for any new children of that generic `radio`. The new `radios` worked because the `channel` property on the children was still owned by the same person who owned the `tune` verb on the parent `radio`.

(Players who had already created child `radios` had to recycle them, however, because their `channel` properties were still owned by themselves. The parent `radio` owner couldn't fix this for them. These players just had to create new `radios` after the parent had been corrected.)

In situations in which a verb is designed to work with a property, the verb and the property need to be owned by the same person. Both the verb and property have the same owner, and they work together, like in the `dates` and `rendezvous` example, mentioned previously.

Exploring an Object

We've mentioned a variety of commands for exploring objects that you come across. This section contains summaries of those commands, and a few more.

`look object`

`@examine` *`object`*

The first and most obvious thing to do with an object is to `look` at it. But you can use the command `@examine` (abbreviated `@exam`) on an object to get a little more detail. `@examine` tells you the object's name, aliases, objnum, who owns it, and its description. `@exam` also tells you the contents of the object (and their objnums), if any.

`@exam` also tells you the obvious verbs on the object. Obvious verbs are ones that are readable and can be invoked by players as command sentences. (Subroutine verbs are not included.)

`look watch`

```
A handsome gold watch with a leather strap, featuring the face of
Lee Kopman. As you inspect the watch more closely, the face of Lee
looks up and whispers, Psst, "Tue Apr 4, 1995 10:21am EDT (3:21pm GMT)".
```

`@exam watch`

```
Lee Kopman watch (#15287) is owned by Spacy (#57882).
Aliases: Lee Kopman watch and watch
A handsome gold watch with a leather strap, featuring the face of
Lee Kopman. As you inspect the watch more closely, the face of Lee
looks up and whispers, Psst, "Tue Apr 4, 1995 10:21am EDT (3:21pm GMT)".
Key: (None.)
Obvious Verbs:
  point <anything> to watch
  @basic_description/@full_name/@short_name watch <anything> <anything>
  @timezone1_standard/@timezone1_daylight/@timezone2_standard/
  @timezone2_daylight    watch <anything> <anything>
  @hour_format/@date_format watch <anything> <anything>
  @timezones watch
  wear/remove watch
  put watch on
  put on watch
  take watch off
  take off watch
  @carried_msg/@worn_msg watch <anything> <anything>
  @says_msg/@whispers_msg watch <anything> <anything>
  read watch
  time <anything> with watch
  g*et/t*ake watch
  d*rop/th*row watch
  gi*ve/ha*nd watch to <anything>
```

The `@contents` command tells you the contents (including the objnums) of an object. You can look inside any object this way, whether or not it's a `$container`.

```
@contents object
```

You use the `@display` command to get information about verbs and properties. After `object`, you can enter a colon (`:`) to get information about verbs, or a period or dot (`.`) to get information about properties. For inherited verbs and properties, use semicolon (`;`) and comma (`,`), respectively. You can combine those, (**`@display something.:;`**, for example) to show the something's properties, verbs, and inherited verbs.

```
@display object
```

You can also get information about a specific verb or property by naming it, such as **`@display something:drop`** to get information about the `drop` verb.

`@display` shows the name of the verb of property, who owns it, and what the permissions are. It also shows property values and verb argument specifiers.

The `@list` command shows you the program for a verb.

```
@list object:verb
```

The `@parents` command shows all the parents of an object, back up to the `Root Class`. This is useful when trying to figure out how to make an object that is like one you have seen, because you can figure out what the appropriate parent object should be.

```
@parents object
```

The `@kids` command shows the children (but just immediate descendants, one level down) of an object.

```
@kids object
```

Some MOOs have additional commands (on `feature` objects) for finding out more information about descendant objects.

eval

The `eval` verb is a handy tool available to programmers that enables you to type in a MOO programming expression (that might appear in the middle of some verb) and see the results immediately.

```
eval 7*6
=> 42
```

You also can use it to examine and alter property values. (Note that `eval` doesn't know the names of objects; you have to use objnums.)

```
eval #2361
=> #2361  (beach ball)
bounce ball
You have to inflate a beach ball before it will bounce!
```

```
eval #2361.inflated
=> 0
eval #2361.inflated = 1
=> 1
eval #2361.inflated
=> 1
inflate ball
It's already inflated.
```

Expressions and Basic Data Types

The MOO language can manipulate several different types of data, such as integer numbers, strings, lists, and, of course, objects. Programmers who are familiar with other computer languages will find the usual arithmetic and logic operators used to form expressions, along with a variety of built-in functions. Learning to write complex programs is beyond the scope of this book, but some of the basics are mentioned, just to give you an idea of what's available. For an in-depth discussion, consult the *LambdaMOO Programmer's Manual* (see Appendix A for a pointer to a WWW version of this).

Expressions

An *expression* is a piece of a program that computes any type of value. `1+1`, for example, is an expression. (The value of that expression is `2`, last time anyone checked.)

When you call built-in functions, they also return a value, and can be used as expressions (or as parts of larger expressions).

Subroutine verbs also can return a value. This is done with the `return` statement,

```
return expression"
```

which causes the verb to finish at that point, giving the value computed by `expression`.

Assignment

Variables are brought into existence simply by naming them in an assignment statement. For example, the statement

```
x = 3;
```

creates a variable *x* and assigns it the value `3`.

The assignment statement `=` also is used to alter the value of a property, as in the following example:

```
this.inflated = 1;
```

Arithmetic

The *arithmetic operators* include + (add), - (subtract), * (multiply), and / (divide). There also is % (remainder). You can use parenthesis to control the precedence of the operators in the usual way.

```
eval 1+2
=> 2
eval (7*6) + (100 - 50)
=> 92
eval 10%3
=> 1
```

The usual comparison operators (>, <, >=, <=) are naturally available. Remember that equals is ==. B. Don't confuse == with the assignment operator =. The `not equals` operator is written !=.

Comparison operators return a `0` (`false`) or a `1` (`true`).

```
eval  1 == 2
=> 0
eval  1 != 2
=> 1
eval  1 < 2
=> 1
```

MOO does not support floating-point numbers (numbers with a decimal point, such as `2.50`); it only supports integers, such as `69`.

Logical Operators

MOO has the usual logical operators familiar to programmers: && (and), ¦¦ (or), and ! (not).

```
eval  1 && 2
=> 1
eval  0 && 2
=> 0
eval  1 ¦¦ 0
=> 1
eval !0
=> 1
eval !(2 > 3)
=> 1
```

Objects Are *false*

An objnum is considered a `false` value in a logical expression.

```
valid(object)
```

In verbs, it's often necessary to check whether a variable holds a `valid` (good) object. For example, an `iobj` might could either be a `valid` object, or could be `$nothing`. The `valid` operator performs this test. `valid` evaluates to `1` (`true`) if `object` is a valid object, or `0` (`false`).

Player Objects

The following determines whether `object` is a player; it returns `1` if it's a player, or `0` if it's some other type of object.

```
is_player(object)
```

Conditional expressions

The following code evaluates the `test-expr`, and if it is `d`, evaluates and returns `then-expr`; otherwise, it evaluates and returns `else-expr`.

```
test-expr ? then-expr¦else-expr
```

The following example computes `1+1`, and if the answer is `2`, returns the string `old math`. If you should happen to find that it returns `new math`, we're all going to be in trouble.

```
eval   ((1+1) == 2) ? "old math" ¦ "new math"
=> "old math"
```

Strings

A *string* is bunch of letters, numbers, spaces, and punctuation. Typically, it is several words or a sentence. Strings are enclosed in quotation marks.

```
eval "this is a string"
=> "this is a string"
player:tell("this is a string")
this is a string
```

If you want to put a quotation mark (") inside your string, you need to precede it with a backslash (\).

```
player:tell("this is a \"string\"")
this is a "string"
```

You can concatenate (append) two strings together with the + operator.

```
eval "this" + "that"
=> "thisthat"
```

To find the length of a string, use the `length` operator.

```
eval length("abcdefd")
=> 7
```

You can use indexing to select just part of a string, either a single character or a range of characters.

```
eval  "abcdefg"[2]
=> "b"
eval "abcdefg"[2..4]
=> "bcd"
eval "abcdefg"[4..length("abcdefg")]
=> "defg"
```

Null strings, such as `""`, are considered `false`.

Lists

MOOs provide for ordered lists. A *list* can contain any kind of MOO data type, such as numbers, strings, objnums, or even other lists.

The following is a list of numbers:

```
{1, 2, 3}
```

The following is a list of objects (`objnums`):

```
{#2361, #1007}
```

The following is a list of lists of objects and strings:

```
{ {#2361, "ball"}, {#1007, "Reader"} }
```

Properties often contain lists of information, which could represent the members of a club, the colors of a rainbow, or a list of tricks that a virtual puppy dog has learned.

Lists can be indexed, spliced together, and so on, in a manner similar to strings.

The `length` function finds out how long a list is.

Empty lists, such as `{}`, are considered `false`.

for Statements

One of the most common things to do with a list is to go through the entire list, repeating some action for every element in the list.

```
for variable in (expression)
  statements to do
endfor
```

The for statement works through the elements of a list, performing the statements to do for every element in expression (which should be a list of some kind).

The following program, for example, counts the number of players and things in the room that you currently are in.

```
number_of_players = 0;
 number_of_things = 0;
 for something in (player.location.contents)
   if (is_player(something))
     number_of_players = number_of_players + 1;
   else
     number_of_things = number_of_things + 1;
   endif
 endfor
 player:tell("There are ",number_of_players," people and ",
             number_of_things," things in here.");
```

This program creates two variables to count the two different types of things as it comes across them; it starts those counts out at zero. You use the for statement to create a variable named something, which contains, one at a time, all the things in the room. (The player has a property location, which is the room the player is in. This in turn has a property contents, which are the things in the room.) The if statement uses the is_player function to test whether each object is a player, and counts them appropriately. When the for is done, a message that tells you the answers prints.

tostr

You use the tostr operator to convert various types of objects, especially numbers.

```
eval tostr(3)
```

```
=> "3"
```

Other Built-In Functions

The commands that you have learned, such as @chparent and @property, actually are verbs (most of which are defined on your player object). These verbs work by using low-level built-in functions with names, such as create, chparent, properties, set_verb_args, and so on.

You use the moveto built-in function to move objects from one location to another. This function keeps the location and contents properties related correctly, so that something doesn't end up inside itself, or inside two different things.

Utility Objects

The MOO includes a collection of objects whose sole purpose is to provide a repository of subroutine verbs that many programmers will find useful to call from their own verbs.

You can get a complete listing of these objects with **`help utilities`**. A few of them are mentioned here.

$string_utils

Because messages are, ultimately, what the MOO is all about, `$string_utils` probably is the most-used utility object.

pronoun_sub

In particular, the `pronoun_sub` verb is commonly used all over the place to make message properties into good messages. Your `beach ball` messages used this facility.

```
$string_utils:pronoun_sub(string)
```

The preceding code line converts a string containing special markers (starting with the `%` character) into a new string with appropriate words substituted. Here are some of the most common substitution markers:

- `%n` is replaced with the name of `player`.
- `%t` is replaced with the name of `this`.
- `%l` is replaced with the name of the location of `player`.
- `%d` is replaced with the name of `dobj`, and `%i` is replaced with the name of `iobj`.

There also are pronouns appropriate for whatever `player` is using the verb at the moment:

- The subject pronoun ("he," "she," "it") is `%s`.
- The object pronoun ("him," "her," "it") is `%o`.
- The possessive pronoun ("his," "her," "its") is `%p`, or for the noun form ("his," "hers," "its") is `%q`.
- The reflexive pronoun ("himself", "herself", "itself") is `%r`.

If you want the substituted words to be capitalized, just capitalize the marker (as in `%P`).

If a player named `yduJ` happened to be in a room called `yduJ's Hairdressing Salon`, for example, a verb with this program

```
player:tell($string_utils:pronoun_sub("%L becomes too popular."));
player:tell($string_utils:pronoun_sub("%N is engulfed in flames."));
```

would produce the following output:

```
yduJ's Hairdressing Salon becomes too popular.
yduJ is engulfed in flames
```

Other, more complicated substitutions, also are available.

Other String Utilities

`english_list` converts a list of strings, such as `{first, second, third}`, into a sentence fragment such as `first, second, and third.`

`title_list` converts a list of objects, such as `{#2361, #1007}`, into a sentence fragment, such as `beach ball and Reader.`

$you

The `say_action` verb on `$you` is another substituting facility that's handy for messages.

The statement

```
$you:say_action("%N %<plays> the piano.");
```

produces

```
You play the piano.
```

on your screen, while everyone else in the room sees

```
Reader plays the piano.
```

In one fell swoop, this amazingly handy utility verb combines pronoun substitution (including action words) and the work that would normally have to be done in two separate statements (a `tell` and an `announce`).

Matching

The `$command_utils`, `$match_utils`, and `$string_utils` objects have verbs for matching word strings with objects, in a way similar to what the server does with each sentence a player types. These utilities are for programs that handle complicated sentences beyond the server's capability.

A verb, for example, might want to handle sentences that include the names of players, regardless of their location on the MOO. The server can't match them unless the players are in the same room as you. But because you know that, (the direct-object is supposed to refer to a player, for example) your verb can be programmed to use `$string_utils:match_player` on `dobjstr` to perform that match yourself.

$object_utils

The `$object_utils` verbs are useful for programs that need to poke around at the MOO world and figure out what types of objects things are, who the parents are, and how things have been programmed.

`$object_utils:isa(`*`object-1, object-2`*`)`

If `object-1` is a descendant of `object-2, then` this verb returns a `1 (true)`. You can use this to see if an object on which the verb is trying to operate is a type of beach ball or hammer, for example.

Other Utilities

`$list_utils` and `$set_utils` have verbs for manipulating lists and sets of things.

`$perm_utils` are for verbs that need to perform sophisticated management of permissions, such as carefully testing whether the player is allowed to use the verb.

Some MOOs have additional utility objects that they've defined. Some MOO's have changed around the way that the utility objects are organized, or details of how those objects are used. Consult the `help` verb and your local wizards for more information.

Containers

Containers are objects that can have other things inside them. They are made out of the `$container` object class.

Children of containers that you might make could represent boxes, sacks, filing cabinets, or anything else that you want to hold something. Containers have verbs for `open` and `close`, and to put things inside and take them back out.

```
@create $container named glass,goblet

You now have glass (aka goblet) with object number #176
and parent generic container (#8).

@describe glass as "A tapered dessert goblet

Description set.

look glass

A tapered dessert goblet.
It is empty.

@create $thing named chocolate mousse,mousse

You now have chocolate mousse (aka mousse) with object number #177
and parent generic thing (#5).

@rename mousse to :chocolate,mousse

Name of #177 (chocolate mousse) is unchanged,
with aliases {"chocolate","mousse"}

@describe mousse as "creamy chocolate mousse that you would love to eat

Description set.

put mousse in glass

glass is closed.

open glass
```

```
You open glass.
```

```
put mousse in glass
```

```
You put chocolate mousse in glass.
```

```
look glass
```

```
A tapered dessert goblet.
Contents:
  chocolate mousse
```

Note that `$root_class` (and therefore, every MOO object) has a `contents` property that could have a list of things in it. For players, the `contents` are the things they are holding (their inventory). The `contents` of a room are the things that are inside the room (players, things, and so on). But these objects are not, strictly speaking, containers, because they are not descended from `$container`; for example, they don't have the `put` and `get` verbs that containers have.

Overriding a Verb

It doesn't make too much sense to `open` and `close` a goblet. You can change the way your container works so that it is always open.

Create a new verb called `close` on your glass.

```
#176:close   this none none
  player:tell("Goblets are always open on top, silly!");
```

This verb *overrides* the `close` verb that was inherited from `$container`. Now if anyone tries to close the glass, they will just see

```
Goblets are always open on top, silly!
```

An Improved Container

You can improve the way the `glass` is described so that you can see the description of its contents. The verb that controls how the glass appears was inherited from `$container`, and is called `look_self`. The following code shows you how to redefine `look self` on your `glass`.

```
#176:look_self   this none this
pass();
if (this.contents)
  player:tell("Inside is ", this.contents[1]:description(),".");
  howmany = length(this.contents);
     if ( howmany > 1)
       player:tell("It also contains:");
       for thing in ( this.contents[ 2..howmany] )
         player:tell("  ", thing:title());
       endfor
     endif
elseif (msg = this:empty_msg())
  player:tell(msg);
endif
```

This is a pretty complicated program and requires some explanation.

The first thing that happens is that you call the built-in function `pass`. `pass` calls your parent's `look_self` verb. The first thing that happens is whatever would normally happen if you had not written a `look_self` for the glass. `pass` is a very powerful feature of the MOO language, enabling you to add behavior to a verb, without duplicating the programming for all the old behavior, as well.

The outermost `if` statement checks to see if anything is inside the glass (if anything is in its `contents` property). If something is inside, the verb prints out `Inside is`, followed by a description of the first thing it finds inside. Notice the call to the `description` verb of whatever is the first `[1]` thing in the `contents`. `description` is a verb that returns a string that tells you what something looks like. Once that is done, you find out how many things are in the `glass`'s `contents` property.

Another `if` statement checks to see if there is more than one thing. If so, the verb continues on with a `for` statement that runs over each of the contents. It starts with the second thing `[2..howmany]` and goes through all of them. For each thing, you print out its name (given by its `title` verb). Some spaces make this indent nicely.

If there wasn't anything in the `glass`, the `elseif` takes over. A variable named `msg` is created and assigned the message that comes from `glass`'s own `empty_msg` verb. Once that is set up, if the variable `msg` is `false`, the verb skips down to the `endif`. Otherwise, it prints out `msg`, which presumably says something about the `glass` being empty.

```
look glass

A tapered dessert goblet.
Inside is creamy chocolate mousse that you would love to eat.
```

Bugs

When a verb doesn't do what it's supposed to, the verb has a *bug*. Bugs are often hard to find.

If you type in program that doesn't work, the server will complain about it, and refuse to install it on the verb. You typically will get an error message about `Parse error`, telling you which line the problem is on.

One of the most common parse errors is forgetting to close all the parenthesis in a statement (the kind that has many parts (of some complicated expression)). Count your parenthesis carefully.

Another common mistake is to leave out the semicolon (`;`) at the end of a statement. A variation on this theme is to add an extraneous semicolon where it doesn't belong (for example, after a keyword like `for` or `endif`).

Runtime Errors

After a verb is programmed, it could blow up with some error message, giving a `backtrace` that tells which line of which verb had the problem. This doesn't mean the line in question is necessarily wrong, but by the time you got to that line, something had gone bad, and the program couldn't proceed any further.

The most common runtime errors are `Variable not found`, `Property not found`, or `Verb not found`. These often indicate a spelling mistake in your program.

Another one is `Type mismatch`, meaning that, for example, you used a number where a string was expected, or a list where an object was expected.

The error `Incorrect number of arguments` may mean that you left out a comma in a call to a built-in function of verb.

Other Problems

A really insidious problem is mistaking `=` for `==`. The former assigns a new value to a variable or a property, while the latter tests to see whether two things are equal. Mixing up those two in your program can have bizarre consequences, even if the verb doesn't actually blow up.

quota and *@audit*

The `@audit` command gives you a list of all of the objects that you've created, tells you where they are presently located, and how much `quota` they are using.

Because the resources of the MOO are not infinite (there's only so much memory and disk space), players are given a quota. `quota` translates roughly into how much disk space on the MOO your objects can use. The important thing to know about `quota` is that it's a personal limit, and you can use yours up. If you run out, you will have to ask whoever is in charge (your local wizard, for example) for more disk space.

The `@quota` command tells you how much disk space you have used up, and how much you have left.

Different MOOs measure `quota` different ways. For example, some MOOs count the number of objects that you've created. Others count exactly how large (in bytes) all your possessions add up to be.

The error message `Object ownership quota exceeded` means that you ran out of `quota`, presumably while trying to `@create` a new object.

@copy

You can copy verbs from one object to another with the `@copy` command; however, this is not recommended. If you think you want to copy a verb, you probably are wrong. Almost always, you should be making the object that you want to copy the verb to a child of some ancestor of the other object. Verbs are meant to be inherited, which does not make a copy. Copying verbs wastes MOO resources.

NOTE Copying verbs also may involve copyright issues. Generally speaking, people own the verbs that they write. If you want to copy something, be sure that you have permission from the author before doing so.

Magic Numbers

Players like numbers that are interesting or easy to remember, but the MOO's `@create` command dishes out whatever number happens to come up next.

Some MOOs have a special object that you can use rather than the `@create` verb, if you would like your object to have a particularly memorable number. For example, it might be a room that you go into and type some incantation like `get 3000 from barrel`. Not all of your objects can have memorable objects, but it's not a bad idea for generic objects that you intend to share with many people.

Magic numbers are mentioned here mostly to remind you that it is considered highly anti-social to sit around doing `@create` and `@recycle`, waiting for a good number to come up. On many MOOs, that sort of behavior will get you thrown off!

Ticks

When a verb is invoked, it is only allowed to execute for a limited period of time, measured in mythical units called *ticks*. If your verb doesn't finish whatever it has to do in its allotted number of ticks, it will blow up. Trying to do something to every player on the MOO is a typical way of running out of ticks. There are techniques for advanced programs to reschedule themselves and get more ticks, but it's often debatable as to whether it's a good idea. The MOO is really designed for executing relatively quick verbs.

Forks

It's possible for a verb to do more than one thing at a time. This works by *forking* off part of the work to be done in parallel.

An example would be a ball that continues to bounce, even after the person who was playing with it has left the room. Another example would be a puppy dog, or perhaps a scary pirate, that wanders around the MOO all the time.

Summary

This chapter has provided you with a short introduction to creating new MOO objects, modifying existing objects, and programming the objects to do what you want. If you want to pursue the topic further, there are several online sources for more information. See Appendix A for pointers.

15

CHAPTER

MUCK AND MUSH PROGRAMMING

By Joseph Poirier

Two types of MUDs that you will find as you explore various MUD environments are MUSHes and MUCKs. Both are TinyMUD-style MUDs that tend to promote social interaction and world-building over combat.

MUCK is a spin-off of TinyMUD, and initially was written by Stephen White. It is based on the concept of MUCKers. MUCKers have programming privileges and are allowed to "muck" around with the database. It has its own programming language, TinyMUF. MUF stands for Multi-User FORTH.

One of the most popular MUCKs is FurryMUCK, where everyone plays anthropomorphic animals (animals with human characteristics).

MUSH originally was written by Lawrence Foard, and stands for *Multi-User Shared Hallucination*. It allows for triggered events and special objects called *puppets*. Many other programmers have since added to the original code. There are several popular MUSHes, including Deep Seas, PernMUSH, and TinyTIM.

This chapter teaches you how to use and program MUSHes and MUCKs. You learn how to set up your character, build rooms, create objects, and program the MUD. Most topics have examples to help you understand the concepts.

This chapter is intended for intermediate MUSH and MUCK users who have connected to a MUSH or MUCK before, experimented with simple commands, perhaps, such as talking and looking, and now want more detail. If you are a beginner, you may want to read the chapter on MUDs, "Interactive Multi-User Realities: MUDs, MOOs, MUCKs, and MUSHes," in an earlier Sams book, *The Internet Unleashed*. That chapter introduces basic MUD concepts more fully, and lists various MUD-related Internet resources you can look into, such as newsgroups, FTP archives, and Frequently-Asked Questions (FAQs).

Finally, throughout this chapter, you will be using a character named `Speedy`.

Thanks

Some portions of this chapter used help files from TinyTIM. Special thanks to Sketch the Cow, one of the original wizards on TinyTIM, for providing this material. TinyTIM is the oldest continually running MUSH still in existence. It was founded in 1990. Its help files are not only useful, they also are amusing to read. You can reach TinyTim at `tim.org`, port `5440`.

Summary of Basic MUSH and MUCK Commands

To refresh your memory, it may be useful to review the basic MUSH and MUCK commands. These basic commands are summarized in Table 15.1.

Table 15.1. Summary of Basic MUSH and MUCK Commands.

Command	*Abbreviation*	*Description*
`drop` *`object`*		Drop the *`object`*
`examine` *`object`*		Obtain detailed *`object`* info
`get` *`object`*		Take the *`object`*
`go` *`direction`*	*`direction`*	Move in the given *`direction`*
`help`		Access help facilities
`home`		Return to your home room
`inventory`	`i`	Inventory
`look`	`l`	Look at room description
`look` *`object`*	`l` *`object`*	Look at *`object`* description
`page` *`player`* `=` *`message`*	`p` *`player=message`*	Page *`player`* with *`message`*

Command	*Abbreviation*	*Description*
`pose` *`message`*	`:`*`message`*	Act out the given `message`
`QUIT`		Quit playing the MUD
`say` *`message`*	`"`*`message`*	Speak the given *`message`*
`take` *`object`*		Take the *`object`*
`whisper` *`player`* `=` *`message`*	`w` *`player`*`=`*`message`*	Whisper *`message`* to *`player`*
`WHO`		See who else is playing

NOTE

Note that the `QUIT` and `WHO` commands are in uppercase.

MUSH and MUCK Basics

There are some basic concepts that you should understand before you begin programming objects on the MUD. In this section, you review basic concepts, such as objects, messages, @commands, actions, locks, substitutions, and other topics.

Basic Objects

MUSHes and MUCKs are composed of four types of things: players, rooms, exits, and objects. *Players* are characters like yourself, all interacting with the MUD at the same time. *Rooms* are the areas that you explore in the MUD. *Exits* connect rooms to one another. *Objects* are comprised of everything else on the MUD. You can create rooms, exits, and objects, and then you can program them to work however you want. That's what this chapter is all about.

Messages

As you play the MUD, your character receives various messages from it. Some of these messages are generated by the MUD from such things as room descriptions, object names, and so on. Other messages are generated by other players. Because events are happening in real-time, you may be sending one message while receiving other messages. You could be talking to someone at the same time that three other people are trying to page you, for example.

Object Numbers

The MUD actually is a big database. Each object has an object number. This number is used to find the object in the database. You can program an object to refer to another object number. In some help systems in a MUD, and in some help guides, the object number is called the `dbref` (database reference).

Money

Most MUDs also have some form of money. This money is needed to create new objects, pay for other objects, build rooms, and so on. You usually can obtain money at random intervals just by wandering around the MUD. This money can be of whatever form the MUD Administrator set up when the MUD was created, such as dollars, pennies, cookies, or new types. If a MUD is based on a particular theme, then the money also may be based on the theme. In the examples in this chapter, *credits* are used as the monetary type.

Home

Your player has a home where it resides. Initially, it will be some common room in the MUD, but you can reset your home to a different room. You can jump to this home room by typing **`home`**.

Killing

Some MUDs allow players to kill each other. Other MUDs frown on this. The command to do this is

```
> kill player = money
```

where the money spent is the percentage chance of success. Spending 100 credits will ensure the other player's demise. Killing something sends it home. The killed player is paid half the amount of money that you spent, as a form of insurance.

@Commands

In addition to the basic commands given earlier, there are many commands that begin with an "at" symbol (`@`). These commands are known, appropriately enough, as "@commands," and they generally indicate that the command operates on the MUD database in some manner.

Actions

Some @commands cause *actions* to occur. For example, you can cause a message to be displayed by using the `emit` command, `@emit` *`message`*. This command displays the message to everyone in the room.

```
> @emit A sudden silence descends upon the crowd.
A sudden silence descends upon the crowd.
```

```
> @emit It's quiet. Too quiet.
It's quiet. Too quiet.
```

Action Lists

You can chain actions together by separating them with semicolons to create an *action list*:

```
> @adrop brick = @emit Hello ; @emit There ; @emit The end
Set.
> drop brick
Hello
There
The end
```

In the preceding example, the list of semicolon-separated `@emit` commands is an action list. When the brick is dropped, the `@adrop` action list is executed.

Attributes

Some @commands set attributes on an object. An *attribute* is a property on an object. For example, the `@drop` attribute sets the message that is displayed when a player drops the object on which it is set:

```
> @drop brick = The brick falls to the ground with a THUD.
Set.
> drop brick
The brick falls to the ground with a THUD.
```

This message appears to the player who dropped the brick.

In addition, you also can set up the brick to display a message to everyone else in the room as well as the player who drops it. To do this, use the `@odrop` attribute, as in the following:

```
> @odrop brick = drops a brick
Set.
```

This particular attribute prepends the output with the name of the player. Thus, if Speedy drops the brick, everyone in the room besides Speedy would see the following:

```
Speedy drops a brick.
```

Finally, on a MUSH, there are attributes that execute action lists when they are triggered, as in the following:

```
> @adrop brick = @emit PLUNK!
Set.
> drop brick
PLUNK!
```

Notice that these attribute commands follow a pattern. The basic attribute has the form `@xxx`, and applies to the player who causes the attribute to be triggered. The second form, `@oxxx`, begins with an `o` and applies to the "others" in the room. The third form, `@axxx`, begins with an `a` and sets the action list associated with the `@xxx` attribute. `@axxx` attributes are available only on MUSHes.

Pronoun Substitutions

Sometimes, when you are programming MUD code, you want to create messages that use the name of the player that is using the object. Or perhaps you want to display a message that depends on the gender of the player using the object. For these occasions, you use *pronoun substitutions*.

Pronoun substitutions use special indicators that begin with a percent sign (%) to substitute appropriate names and subject forms. For example, the %N pronoun substitution substitutes the player's name for the %N indicator:

```
> @emit %N is here! Look at %o!
Speedy is here! Look at him.
```

Each indicator has two types: uppercase and lowercase. Uppercase indicators, such as the %N in the preceding example, substitute the appropriate pronoun with its first letter in uppercase. Lowercase indicators, such as %o, leave the pronoun or name as it is—so a name that begins with a lowercase letter would remain lowercase.

As a handy reference, Table 15.2 shows the different pronoun substitutions that you can use in your messages and actions.

Table 15.2. Pronoun Substitutions for MUSHes and MUCKs.

Indicator	*Pronoun Substitution*	*Examples*
%N, %n	Player's name	Speedy, Speedy
%S, %s	Subjective form	He/She/It/They, he/she/it/they
%O, %o	Objective form	Him/Her/It/Them, him/her/it/them
%P, %p	Possessive form	His/Her/Its/Their, his/her/its/their
%A, %a	Absolute possessive form	His/Hers/Its/Theirs, his/hers/its/theirs

Locks

Locks stop players or objects from performing certain actions on other players or objects. When you create a lock on an object, you are limiting how that object can be used.

When you lock an object, you also specify which key can be used to get past the lock. The key can be another object, or a player, or a combination of objects and players. You can program a lock to check to see if the player attempting to pass the lock is either the key itself or is carrying the key.

Following is the format of the lock command:

```
> @lock object = key
```

When you pass an object's lock, you see the `@success` attribute of the object, and the other players in the room see the `@osuccess` attribute. Additionally, the `@asuccess` attribute is triggered in MUSHes.

When you fail to pass an object's lock, you see the `@fail` attribute of the object, and the other players in the room see the `@ofail` attribute. On a MUSH, the `@afail` attribute also is triggered.

Table 15.3 shows some special characters you can use to specify the way in which locks are passed.

Table 15.3. Specifying locks with special characters.

Character	*Meaning*
&	Boolean AND
¦	Boolean OR
!	Boolean NOT
#	`Database Reference`
*	`Player Named`
=	`Is`
+	`Is Carrying`
–	`Indirect`

NOTE The = and + lock special characters apply only to MUSHes.

To see how to program locks, look at some examples. First, look at an example of the simplest lock. If `Speedy` is object number `#1111`, and you want only `Speedy` to be able to go through a special doorway, you could type the following:

```
> @lock doorway = #1111
```

You also could program this using the `Player Named` directive.

```
> @lock doorway = *Speedy
```

Or, if you want to program a treasure chest to need a skeleton key, you could specify that the object is the key. Assume that the skeleton key is object number `#2222`.

```
> @lock chest==#2222¦+#2222
```

Or, perhaps you want to lock a platform to accept either a red gem with object `#555` or a blue gem with object `#666`, as in the following:

```
> @lock platform = #555 ¦ #666
```

As another example, perhaps you want to program a scepter so that it only works if you do *not* have a Kryptonite object around. If the Kryptonite was object #3333, then

```
> @lock scepter = ! #3333
```

would lock it against the Kryptonite.

There are a few specialized forms of locks that are available on MUSHes. They are shown in Table 15.4.

Table 15.4. Lock types.

Lock Type	*Syntax*
Default Lock	`@lock/default object=key`
Enter Lock	`@lock/enter object=key`
Give Lock	`@lock/give object=key`
Leave Lock	`@lock/leave object=key`
Link Lock	`@lock/leave object=key`
Page Lock	`@lock/page object=key`
Tport Lock	`@lock/page object=key`
Use Lock	`@lock/use object=key`

Default locks are the locks that have been discussed so far. *Enter* locks determine who or what can enter an object. *Give* locks determine who may give the object. *Leave* locks determine who or what can leave an object. *Link* locks determine what can link to the object. *Page* locks determine who or what can page an object. *Tport* locks determine who may teleport to the object if the object's LINK_OK flag is set. *Use* locks determine who or what can use an object.

Only MUSHes have lock types other than default locks. For example, if you were on a MUSH and you wanted to stop the character Goober from paging you, you could type

```
> @lock/page me=! *Goober
```

But wait! There's more! You also can create indirect locks on MUSHes. These locks depend on something else's lock. To create them, put the @ symbol in front of the lock. For example,

```
> @lock portal = @#999
```

means "set the portal's lock to be the same as object #999's lock."

Finally, you can specify certain attributes for locks. These attributes must be ones that can be read on another player, such as @sex or @name. You can use asterisks (*) as wildcard characters for pattern-matching of the attribute values. So, to lock a door so that only men can go through, use the following:

```
> @lock door = sex:male
```

To lock a barn so that only objects with `horse` in their descriptions can pass, you would use the following:

```
> @lock barn = desc:*horse*
```

Flags

Objects may have certain flags. *Flags* are used to set characteristics of an either/or nature on objects, such as setting a flag to indicate that an object is or is not invisible. Various flags are discussed later in this chapter, but a general explanation of the syntax is useful at this point.

The syntax for turning a flag `ON` is

```
> @set object = flag
```

and the syntax for turning it `OFF` is

```
> @set object = ! flag
```

Teleporting

One special movement command that is available on MUDs is the `teleport` command, which instantaneously moves you to a destination room, as in the following:

```
> @tel object = room
```

So, if room `#333` was a Grand Ballroom, and you wanted to go there without wandering through all the rooms in between, you could type

```
> @tel me = #333
```

NOTE

On some MUDs, this command may be turned off.

You can only teleport to rooms that you own or rooms that have their `JUMP_OK` flag set.

How to Program Your Player

To learn how to program your player, begin by looking at the commands and attributes that apply to a player. In the following examples, recall that you are using a character named `Speedy`.

Once you obtain a new character on a MUD, one of the first things you should do is change your password, as in the following:

```
> @password old-password = new-password
```

Keeping your password secret is just as important on a MUD as it is on a computer system. If someone gains access to your character, they can impersonate you. They can harass other players, say obscene things, and perhaps even get your character removed from the MUD. So it is a good idea to change your password initially, and from time to time, to make sure no one gains illegal access to your character.

Next, set the description for your character. Following is the syntax for the `@describe` command:

```
> @describe object = description
```

You can set the description for your example character, `Speedy`, with the following syntax:

```
> @describe me = You see a whirling, blurry, blue tuft ball of fur and bare feet!
> look me
You see a whirling, blurry, blue tuft ball of fur and bare feet!
```

Next, set the gender of your character with the `@sex` command, as in the following syntax:

```
> @sex me = male
Set.
```

The `@sex` attribute setting is based on the first letter of its setting. If the first letter is `M` or `m`, for example, gender is considered to be male. If the first letter is `F`, `f`, `W` or `w`, then gender is female. On some MUDs, if the first letter is `P` or `p`, the gender will be plural. Any other values of the first letter will set the gender neuter.

You also should lock your character so that you can't be robbed or picked up by another character:

```
> @lock me = me
Set.
```

The special term `me` stands for your character.

How to Program Rooms

Players are not the only things on a MUD. A MUD also has many *rooms* for those players to explore. These rooms give the MUD atmosphere and an identity. You can create your own rooms on a MUD, too. You then can reset your home to one of the rooms you have created.

Rooms are connected to one another by *exits*. You also can create exits and program them to enhance the atmosphere of the rooms you are creating.

In this portion of the chapter, you will be creating several rooms, writing the descriptions for them, linking them together by creating exits, and adding various other enhancements to create a little scene. As you go through the examples, you will learn the commands to build your own rooms. The portion of the MUD that you eventually create is limited only by your imagination.

This example will be a swimming pool with a few diving boards above it. You will be able to jump into the pool and dive into it deeper. You will be able to jump off the diving boards into the pool, too.

Begin by visualizing the pool scene (see Figure 15.1). Once you have an idea of what you want to create, you can generalize it into the actual rooms and exits that you will need to create. This is shown in Figure 15.2. Drawing out the system of rooms in this manner helps you get the exits from room to room correct, since they should make topological sense.

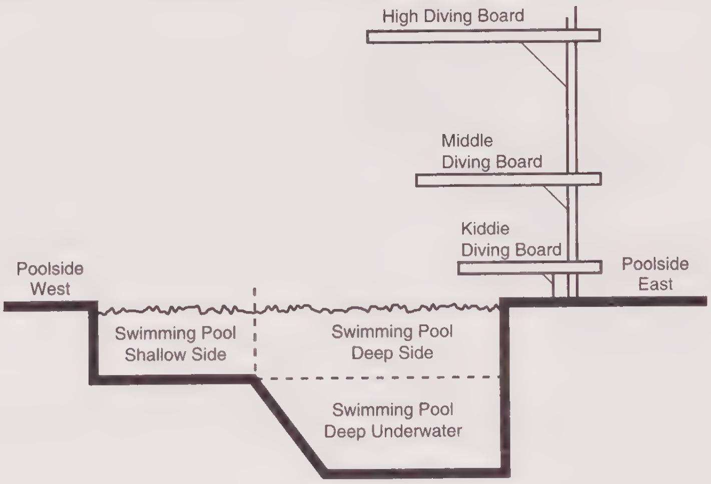


Figure 15.1.
This is the visualization of the environment.

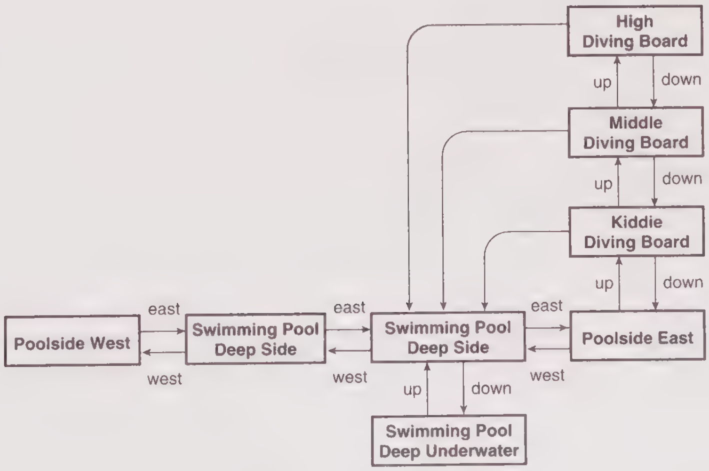


Figure 15.2.
Here are the rooms and exits of the environment.

Begin by constructing the pool. The pool itself is composed of three rooms. One room, on the western side, depicts the shallow end of the pool. The eastern side of the pool is the deep side, and is created using two rooms. The top room represents the surface of the deep side, and the bottom room is the underwater portion of the deep end of the pool. The command to create a room is `@dig`.

```
> @dig room-name
```

When you type this, a room with the given *`room-name`* is created. You also will be told the object number of the new room.

First, create the three rooms that comprise the pool:

```
> @dig Swimming Pool — Shallow Side
Swimming Pool—Shallow Side created with room number #1200.
> @dig Swimming Pool — Deep Side
Swimming Pool—Deep Side created with room number #1201.
> @dig Swimming Pool — Deep Underwater
Swimming Pool—Deep Underwater created with room number #1202.
```

You then can write descriptions for each of the rooms. First, go to the room you want to describe, and then set the room description by using the special room-name `here`, which stands for the room that you are in. Note that your room numbers may be different from the ones in these examples:

```
> @tel me=#1200
Swimming Pool—Shallow Side(#1200R)
> @desc here=You are floating in the shallow side of the pool. The pool gets deeper
to the east. You can see the blue and white tiles on the bottom of the pool clearly.
Set.
> @tel me=#1201
Swimming Pool—Deep Side (#1201R)
> @desc here=You are bobbing up and down in the deep side of the pool. The pool gets
shallow to the west. A diving platform rises above the water, to the east. You can
see the blue and white tiles on the bottom of the pool faintly through the shimmering
water below you.
Set.
> @tel me=#1202
Swimming Pool—Deep Underwater (#1202R)
> @desc here=You are swimming underneath the surface of the deep side of the pool. A
diving platform glints through the sparkling waters above you. You can see the blue
and white tiles on the bottom of the pool clearly.
Set.
```

NOTE

It is assumed that teleportation is turned on in these examples.

You can't simply walk into these rooms yet because you not have linked them to the rest of the MUD using exits. So, for now, these rooms are not connected to anything. They are floating in hyperspace, so to speak.

One way to connect your rooms to the rest of the MUD is to wander around and find a hotel, apartment building, or similar place where you can get your own room. By following the instructions at a hotel lobby, for example, a personal room will be created for you. You then can link the other rooms that you create to your new hotel room by creating exits between them.

How to Program Exits

Next, you can connect these rooms to one another by creating *exits* between them. You can create exits by using either a special form of the `@dig` command, or by using the `@open` command. Look at the syntax of the `@open` command, as follows:

```
> @open in1;in2;in3;... = #other-room , out1;out2;out3;...
```

The `in1;in2;in3;...` portion is called the *in-list*, and the `out1;out2;out3;...` portion is called the *out-list*. An exit from the current room to the room specified by the given `other-room` number will be created. This exit will have the names specified in the in-list. Another exit also will be created, from the other room back to the current room, and will be named according to the out-list (see Figure 15.3).

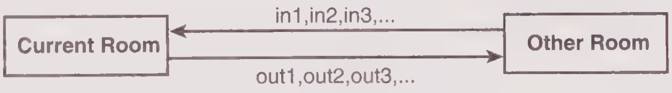


Figure 15.3.
The in-list and out-list of the `@open` *command.*

Each name in the in-list is an alias for the exit from the current room to the other room, and each name in the out-list is an alias for the exit from the other room back to the current room.

If you are on a MUSH, the first alias in the in-list, `in1`, is used in the `Obvious Exits` portion of the current room description. Likewise, the first alias in the out-list, `out1`, is used in the `Obvious Exits` portion of the other room description. Also, some MUDs do not allow you to specify the out-list at the same time you construct the in-list. On these MUDs, you can simply create the exits using two `@open` commands.

You must own the room from which you are trying to create an exit. You also must own the room that you are linking to, or that room must be set `LINK_OK`.

You can leave off the out-list if you want. In that case, the exit from the other room to the current room would not be created.

Using the `@open` commands, create the exits between the rooms that you have created. Name the exits according to the diagram given previously in Figure 15.2.

First, you need to go to the shallow side of the pool.

```
> @tel me = #1200
Swimming Pool—Shallow Side (#1200R)
You are floating in the shallow side of the pool. The pool gets deeper to the east.
You can see the blue and white tiles on the bottom of the pool clearly.
```

This room now is your current room. You create an exit from this room to the deep side, and another exit from the deep side back to this room, as in the following:

```
> @open east;e;swim = #1201 , west;w;swim
Opened.
Linked.
Opened.
Linked.
```

The `Opened` and `Linked` messages indicate that the two exits are being created and linked. These two exits also have their own object numbers.

Now you can move to the deep side by going east!

```
> east
Swimming Pool—Deep Side (#1201R)
You are bobbing up and down in the deep side of the pool. The pool gets shallow to
the west. A diving platform rises above the water, to the east. You can see the blue
and white tiles on the bottom of the pool faintly through the shimmering water below
you.
Obvious exits:
west
```

Note that the `Obvious Exits` portion of the room description does not appear on MUCKs.

Next, create the exits from this room to the underwater portion of the deep side, as in the following:

```
> @open down;d;dive;bottom = #1202 , up;u;rise;surface
Opened.
Linked.
Opened.
Linked.
```

Now you can go to the bottom of the deep end, and back up.

```
> down
Swimming Pool—Deep Underwater (#1202R)
You are swimming underneath the surface of the deep side of the pool. A diving
platform glints through the sparkling waters above you. You can see the blue and
white tiles on the bottom of the pool clearly.
Obvious exits:
up
> up
Swimming Pool—Deep Side (#1201R)
You are bobbing up and down in the deep side of the pool. The pool gets shallow to
the west. A diving platform rises above the water, to the east. You can see the blue
and white tiles on the bottom of the pool faintly through the shimmering water below
you.
Obvious exits:
west down
```

Now that the swimming pool is complete, you can build the diving platform structure. To do this, you can use the form of the `@dig` command that also specifies exits:

```
> @dig other-room-name = in1;in2;in3;... , out1;out2;out3;...
```

This form of the `@dig` command creates a room whose name is specified by the *`other-room-name`*, and it also creates two exits. One exit goes from the current room to the other room, and has the aliases given by the in-list. The other exit goes from the other room to the current room and has the aliases given by the out-list.

This command is not available on MUCKs; instead, if you are on a MUCK, `@dig` the room and then link the exits using the `@open` command.

Using the `@dig` command, you can create the diving platform. Assume that the surface of the deep side of the pool is the current room.

First, create the east side of the pool:

```
> @dig Poolside East = east;e;out , west;w;in;swim;pool
Poolside East created with room number #1207.
Opened.
Linked.
Opened.
Linked.
> east
Poolside East (#1207R)
Obvious exits:
west
> @desc here = You are standing on the east side of the pool, near the deep end. The
pool is to your west. A diving platform rises above you here, reachable by a ladder.
Set.
```

Then create the diving platform, which is really three different rooms:

```
> @dig Kiddie Diving Board = up;u;ladder;board;platform;kiddie , down;d
Kiddie Diving Board created with room number #1210.
Opened.
Linked.
Opened.
Linked.
> up
Kiddie Diving Board (#1210R)
Obvious exits:
down
> @desc here = You are on the kiddie diving platform. The end of the platform is only
a few feet above the surface of the deep end of the pool. You can see higher diving
platforms above you. A ladder leads up to higher platforms, and down to the poolside.
Set.
> @dig Middle Diving Board = up;u;middle , down;d;kiddie
Middle Diving Board created with room number #1213.
Opened.
Linked.
Opened.
Linked.
> up
Middle Diving Board (#1213R)
Obvious exits:
down
> @desc here = You are on the middle diving platform, which hovers about twelve feet
above the surface of the deep end of the pool. Below you is the kiddie diving plat-
form, and above you is the ominous high diving platform. A ladder leads up to the
high platform, and down to the kiddie platform.
Set.
> @dig High Diving Platform = up;u;high , down;d;middle
High Diving Board created with room number #1216.
Opened.
```

```
Linked.
Opened.
Linked.
> up
High Diving Board (#1216R)
Obvious exits:
down
> @desc here = You are standing cautiously on the highest diving platform! The pool
is about forty feet below you, and looks awfully far away! Other diving platforms are
below you, reachable by a ladder leading down.
Set.
```

Now that the diving platform is set up, you can create the links from each diving board down into the deep end of the pool. Currently, you are on the highest diving board. The deep side of the pool was room #1201 in this example.

```
> @open dive;jump;west;w = #1201
Opened.
Linked.
> down
Middle Diving Board (#1213R)
You are on the middle diving platform, which hovers about twelve feet above the
surface of the deep end of the pool. Below you is the kiddie diving platform, and
above you is the ominous high diving platform. A ladder leads up to the high plat-
form, and down to the kiddie platform.
Obvious exits:
down up
> @open dive;jump;west;w = #1201
Opened.
Linked.
> down
Kiddie Diving Board (#1210R)
You are on the kiddie diving platform. The end of the platform is only a few feet
above the surface of the deep end of the pool. You can see higher diving platforms
above you. A ladder leads up to higher platforms, and down to the poolside.
Obvious exits:
up down
> @open dive;jump;west;w = #1201
Opened.
Linked.
```

Note that these particular @open commands did *not* specify an out-list. That is because you only want to create exits from the diving boards to the pool. You don't want to be able to somehow instantly go from the pool to the highest diving board!

Finally, create the west side of the pool, near the shallow end. You can get to the shallow end of the pool by either moving there, now that all the exits between rooms have been created, or by using teleportation.

```
> @tel me = #1200
Swimming Pool—Shallow Side (#1200R)
You are floating in the shallow side of the pool. The pool gets deeper to the east.
You can see the blue and white tiles on the bottom of the pool clearly.
Obvious exits:
east
> @dig Poolside West west;w;out , east;e;in;swim;pool
Poolside West created with room number #1220.
Opened.
```

```
Linked.
Opened.
Linked.
> west
Poolside West (#1220R)
Obvious exits:
> @desc here = You are standing on the west side of the pool, near the shallow end.
The pool is to your east, and you can see a diving platform on the other side of the
pool.
Set.
```

NULL Exits

One way to make your rooms better is to display helpful messages to people when they try to move in a direction that doesn't exist in that room.

Usually, when a player tries to move in an invalid direction (a direction for which there is no exit of that name), the MUD displays a default `Huh? (Type 'help' for help)` message. A more user-friendly message might be `You cannot go in that direction`. To display such a message, you can create a NULL exit.

To create a NULL exit, you first create an exit on the room and name it the various invalid directions in that room. Lock the exit to `#0`, and set it `DARK`. Finally, set its `@fail` message to the message that you want to appear when a player tries to go in that direction.

When someone tries to go in an invalid direction, the `@fail` message will be displayed to them, because the exit is locked against them. The exit is set `DARK` so that it will not appear in `Obvious Exits` or similar descriptions.

As an example, consider the top of the diving board. You can go west and fall into the pool, or you can go down the ladder, but you should not be able to go north, south, east, or up. You can create a NULL exit to display a better message, as in the following:

```
> @tel me = #1216
High Diving Board (#1216R)
You are standing cautiously on the highest diving platform! The pool is about forty
feet below you, and looks awfully far away! Other diving platforms are below you,
reachable by a ladder leading down.
Obvious exits:
down dive
> @open north;n;south;e;east;e;up;u = here
Opened.
Linked.
> @lock north = #0
Locked.
> @set north = DARK
Set.
> @fail north = You cannot go in that direction.
Set.
```

Now that the NULL exit is set up, you can test it out, as in the following:

```
> north
You cannot go in that direction.
```

```
> south
You cannot go in that direction.
> east
You cannot go in that direction.
> up
You cannot go in that direction.
```

NULL exits on the other rooms in this example can be set similarly.

Special Exits

You also can create special exits that display special messages that are similar to NULL exits, as in the following:

```
> @open bounce
Opened.
Linked.
> @ lock bounce = #0
Locked.
> @set bounce = DARK
Set.
> @fail bounce = You bounce up and down on the diving board. BOING!
Set.
```

Now, when you type `bounce`, you see the following message:

```
> bounce
You bounce up and down on the diving board. BOING!
```

Enhancing Exits

You can define several attributes on exits so that messages are displayed to players who go through them, as well as to people who are in the rooms that the player is entering and leaving. By setting the exit attributes on your exits, you can add to the atmosphere of the MUD.

There are three main types of exit attributes. The `@succ` attribute message is displayed to a player when that player goes through the exit successfully. The `@osucc` attribute message is displayed to the other players in the room that the player is leaving when that player goes through the exit successfully. The `@asucc` attribute causes an action list to be performed when a player leaves a room successfully.

NOTE The `@asucc` attribute is not available on MUCKs.

For the other side of the exit, the `@odrop` attribute message is displayed to the other people in the room that the player is entering. The `@adrop` attribute causes an action list to be performed when a player enters a room successfully.

NOTE

The `@adrop` attribute does not exist on MUCKs.

The `@fail` attribute message is displayed to a player when that player uses an exit unsuccessfully. The `@ofail` attribute is displayed to other people in the room that a player fails to leave. The `@afail` attribute causes an action list to be performed when an exit is used unsuccessfully.

NOTE

The `@afail` attribute is not available on MUCKs.

Also, note that the `@oxxx` attributes (`@osucc`, `@odrop`, and `@ofail`) automatically prepend the name of the triggering player to their messages.

To illustrate how these exit attributes are used, define some exit attributes for your example. You can use the `examine` command to find the object numbers of the exits in the room. You can start on the shallow end of the pool.

```
> examine here
Swimming Pool—Shallow Side (#1200RF)
Type: ROOM Flags: FLOATING
You are floating in the shallow side of the pool. The pool gets deeper to the east.
You can see the blue and white tiles on the bottom of the pool clearly.
Owner: Speedy Key: *UNLOCKED*
Contents:
Speedy(#1111Pc)
Exits:
north;n;south;s;up;u;down;d(#1230ED)
west;w;out(#1221E)
east;e;swim(#1203E)
```

Note the exits displayed at the bottom of the preceding code. There are three exits mentioned. The `east` and `west` exits allow the player to move into and out of the pool, respectively. You created those exits in your earlier examples. The `north/south/up/down` exit is a NULL exit that displays a `You cannot go in that direction message to the player`. The `D` flag indicates that it is dark.

You then can write some exit attributes on the east and west exits:

```
> @succ east = You swim to the deep end of the pool.
> @osucc east = swims to the deep end of the pool.
> @odrop east = swims over to this side of the pool.
> @fail east = You try to swim to the deep end of the pool, but cannot.
> @ofail east = tries unsuccessfully to swim to the deep end of the pool.
> @succ west = You climb out of the pool, dripping wet.
> @osucc west = climbs out of the pool.
> @odrop west = climbs out of the pool.
> @fail west = You try to climb out of the pool, but find you cannot.
> @ofail west = tries to climb out of the pool, but cannot.
```

You then can set up the exits in the deep end of the pool. Note how the new `@succ` message is now displayed when you move to the deep end.

```
> east
You swim to the deep end of the pool.
Swimming Pool—Deep Side (#1201R)
You are bobbing up and down in the deep side of the pool. The pool gets shallow to
the west. A diving platform rises above the water, to the east. You can see the blue
and white tiles on the bottom of the pool faintly through the shimmering water below
you.
Obvious exits:
west east down
```

Now you can set up the exit attributes from this room:

```
> @succ west = You swim to the shallow side of the pool.
> @osucc west = swims to the shallow side of the pool.
> @odrop west = swims over to this side of the pool.
> @fail west = You try to swim to the shallow end, but cannot.
> @ofail west = tries to swim to the shallow end, but cannot.
> @succ east = You climb out of the pool, dripping wet.
> @osucc east = climbs out of the pool.
> @odrop east = climbs out of the pool.
> @fail east = You try to climb out of the pool, but find you cannot.
> @ofail east = tries to climb out of the pool, but cannot.
> @succ down = You dive deeper into the pool.
> @osucc down = dives deeper into the pool.
> @odrop down = suddenly swims into view.
> @fail down = You try to dive deeper into the pool, but find you cannot.
> @ofail down = tries to dive deeper into the pool, but finds %s cannot.
```

Finally, you can set up the exit attributes for the deep portion of the deep side.

```
> down
You dive deeper into the pool.
Swimming Pool—Deep Underwater (#1202R)
You are swimming underneath the surface of the deep side of the pool. A diving
platform glints through the sparkling waters above you. You can see the blue and
white tiles on the bottom of the pool clearly.
Obvious exits:
up
```

Following are the commands to set up the exit from here to the surface.

```
> @succ up = You swim back up to the surface of the pool.
> @osucc up = swims back up to the surface.
> @odrop up = returns from the bottom of the pool.
> @fail up = You try to return to the surface, but cannot.
> @ofail up = tries to return to the surface, but cannot.
```

That sets up the pool exits. You can program the diving board exits similarly.

Lists of Exits

You can obtain a list of the exits leading out of a room by using the `examine` command on the room:

```
> examine here
Swimming Pool—Shallow Side (#1200RF)
```

```
Type: ROOM Flags: FLOATING
You are floating in the shallow side of the pool. The pool gets deeper to the east.
You can see the blue and white tiles on the bottom of the pool clearly.
Owner: Speedy Key: *UNLOCKED*
Contents:
Speedy(#1111Pc)
Exits:
north;n;south;s;up;u;down;d(#1230ED)
west;w;out(#1221E)
east;e;swim(#1203E)
```

You can get a list of the exits leading into a room by using the `@entrances` command, as in the following:

```
> @entrances here
Swimming Pool—Deep Side(#1201R) (west;w;swim(#1203E))
Swimming Pool—Shallow Side(#1200R) (north;n;south;s;down;d;up;u(#1230ED))
Poolside West(#1220R) (east;e(#1221E))
```

Removing Exits

You can break an exit by unlinking it from a destination room, as in the following:

```
> @unlink exit
```

The `unlink` command unlinks the exit from its destination room. You then could relink it to a different destination room; however, you should do this soon after unlinking it because unlinked exits can be stolen by other players and relinked to rooms of their choosing. Exits become owned by the player that links them.

To actually delete the link, use the `@destroy` command on MUSHes or the `@recycle` command on MUCKs.

Creating Objects

Next, you will learn how to create objects in a MUD. You can create any type of object you want, and you can describe it however you wish.

To create an object, you use the `@create` command, as in the following:

```
> @create object-name = cost
```

The *cost* you give is how much the object is worth if it is destroyed or recycled. When you create an object, the MUD will make sure you can pay the cost, and then display a message that tells you that the object has been created with a certain object number. If you omit the cost, the default cost is 10.

Once the object has been created, you can refer to enough of its name to differentiate it from other objects. The MUD will perform pattern matching to determine which object is being referred to.

TIP

If you ever want to rename the object you have created, you can use the `name` command, as in the following:

```
> @name old-name = new-name
```

For your swimming pool example, you can create a rubber duck to float around in the pool.

```
> @create rubber duck
rubber duck created with object number #3000.
> @desc rubber duck = This is a large, plastic, inflated rubber duck, about two feet
across. It has a large yellow bill, and amusing red polka dots painted on it.
> drop duck
Dropped.
```

You then can set the home of the object to the deep end of the pool so that it will go back there if it is ever sent home:

```
> @link rubber duck = #1201
Home set.
```

Locks on Exits

Now you are going to program the MUD so that a player cannot take the rubber duck underwater—the assumption being that it is inflated, and won't go underwater. You can program the exit between the Deep Side room and the Deep Underwater room—the down exit—to prevent the rubber duck from passing through. Remember that the rubber duck is object `#3000`

```
> @lock down=! #3000
```

which states that an object can go down if it is not object `#3000`.

Once this lock is set, the `@fail` and `@ofail` attributes will be displayed if a player tries to take the rubber duck down into the deep portion of the deep end of the pool:

```
> look
Swimming Pool—Deep Side (#1201R)
You are bobbing up and down in the deep side of the pool. The pool gets shallow to
the west. A diving platform rises above the water, to the east. You can see the blue
and white tiles on the bottom of the pool faintly through the shimmering water below
you.
Contents:
rubber duck (#3000)
Obvious exits:
west east down
> take duck
Taken.
> down
You try to dive deeper into the pool, but find you cannot.
```

Container Objects

You can create container objects on MUSHes. To create an inflated floating raft that players can hop onto, you can program the raft to be a container. Containers can hold other objects inside them. You can turn objects into containers by setting them `ENTER_OK`:

```
> @create floating raft
floating raft created with object number #7500.
> @desc raft = You see an inflated, floating raft.
Set.
> @set raft = ENTER_OK
Flag set.
```

The raft is now a container.

Container Attributes

Containers have special attributes that are displayed to players that enter or leave the object, as well as a special description that is displayed to players inside the container.

The `@desc` attribute is displayed to players who look at the container from the outside. The `@idesc` attribute is shown to players who look at the container from the inside, as in the following:

```
> @idesc raft = You are on top of an inflated, floating raft in the pool.
```

The `@enter` attribute is displayed to a player that is entering a container. The `@oenter` attribute is displayed to players that are already inside the container when an object enters it, and the `@oxenter` attribute is displayed to players outside the container when an object enters the container. The `@leave`, `@oleave`, and `@oxleave` attributes correspond to objects leaving a container in the same manner.

```
> @enter raft = You jump onto the floating raft!
Set.
> @oenter raft = jumps onto the floating raft with you.
Set.
> @oxenter raft = jumps onto the floating raft.
Set.
> @leave raft = You jump off of the floating raft.
Set.
> @oleave raft = jumps off of the floating raft.
Set.
> @oxleave raft = jumps off the floating raft.
Set.
```

Now you can test it out. To enter and leave objects, you use the `enter` and `leave` commands, respectively:

```
> enter raft
You jump onto the floating raft!
```

```
floating raft (#7500e)
You are on top of an inflated, floating raft in the pool.
> leave
Swimming Pool—Deep Side (#1201R)
You are bobbing up and down in the deep side of the pool. The pool gets shallow to
the west. A diving platform rises above the water, to the east. You can see the blue
and white tiles on the bottom of the pool faintly through the shimmering water below
you.
Contents:
floating raft (#7500e)
rubber duck (#3000)
Obvious exits:
west east down
```

Drop-To Rooms

You also can use the `@link` command to set up a `drop-to` room. If you are in a `drop-to` room and you drop an object, the object actually falls into some other room. Following is the syntax to set up a `drop-to` room:

```
> @link drop-to-room = #other-room
```

In your diving board example, you could set up the various diving boards to be `drop-to` rooms. Anything that is dropped in them would fall down to the Poolside East room instead. (Recall that Poolside East was room `#1207`.)

Go to the Kiddie Diving Board, which is room `#1210` in your example:

```
> @tel me = #1210
Kiddie Diving Board (#1210R)
You are on the kiddie diving platform. The end of the platform is only a few feet
above the surface of the deep end of the pool. You can see higher diving platforms
above you. A ladder leads up to higher platforms, and down to the poolside.
Obvious exits:
up down dive
> @link here = #1207
Dropto set.
> up
Middle Diving Board (#1213R)
You are on the middle diving platform, which hovers about twelve feet above the
surface of the deep end of the pool. Below you is the kiddie diving platform, and
above you is the ominous high diving platform. A ladder leads up to the high plat-
form, and down to the kiddie platform.
Obvious exits:
down up dive
> @link here = #1207
Dropto set.
> up
High Diving Board (#1216R)
You are standing cautiously on the highest diving platform! The pool is about forty
feet below you, and looks awfully far away! Other diving platforms are below you,
reachable by a ladder leading down.
Obvious exits:
```

```
down dive
> @link here = #1207
Dropto set.
```

Now, if you you're carrying the rubber duck, and you dropped it from the High Diving Board, it would fall to the Poolside East room:

```
> drop duck
Dropped.
> down
Middle Diving Board (#1213R)
You are on the middle diving platform, which hovers about twelve feet above the
surface of the deep end of the pool. Below you is the kiddie diving platform, and
above you is the ominous high diving platform. A ladder leads up to the high plat-
form, and down to the kiddie platform.
Obvious exits:
down up dive
> down
Kiddie Diving Board (#1210R)
You are on the kiddie diving platform. The end of the platform is only a few feet
above the surface of the deep end of the pool. You can see higher diving platforms
above you. A ladder leads up to higher platforms, and down to the poolside.
Obvious exits:
up down dive
> down
Poolside East (#1207R)
You are standing on the east side of the pool, near the deep end. The pool is to your
west. A diving platform rises above you here, reachable by a ladder.
Contents:
rubber duck (#3000)
Obvious exits:
up west
```

Finally, if you set a `drop-to` room to also be `STICKY`, then the objects that are dropped in it will not fall to the other room until everyone has left the `drop-to` room. This creates a delayed `drop-to` effect.

Percent Substitutions

Now turn to things that are specific to MUSHes and are *not* found in MUCKs.

In MUSHes, there is a special storage area that holds various values whenever an action occurs. You can access these values using percent substitutions. *Percent substitutions* are simply variables that start with a percent sign. The pronoun substitutions mentioned at the beginning of this chapter actually are types of percent substitutions.

One of the percent substitutions is `%N`. When an action occurs, the `%N` variable holds the name of the object that caused the action to occur. Along with that, the object number of the object causing the action is held in the `%#` variable. If the action affects another object, the object number of the affected object can be found in the `%!` variable. The location where the action is occurring is held in the `%l` variable. Table 15.5 has a list for easy reference.

Table 15.5. Special percent substitutions.

Substitution	*Meaning*
`%#`	Object number of object causing an action
`%!`	Object number of object being acted upon
`%l`	Object number of location where action is occurring

For example, if `Speedy` is object `#1111`, and he picks up a `brick` whose object number is `#4444`, then `%N` would hold `Speedy`, `%#` would be `#1111`, and `%!` would hold `#4444`. If `Speedy` picked up the `brick` while in room `#7777`, then `%l` would be `#7777`.

You then could use these percent substitutions in your own MUSH programming code.

Formatting Codes

There are a few percent substitutions that cause special formatting to occur. You can use these special codes in your descriptions and other messages when you need to format them in a special way. Table 15.6 shows a list of these codes.

Table 15.6. Special Text Formatting Codes

Substitution	*Format*
`%r`	Carriage return and new line
`%t`	Tab
`%b`	Blank space
`%%`	Percent sign (`%`)

You can use these formatting codes in your code. The tab character (`%t`), for example, can be useful when formatting tables or lists of information.

Registers and Triggers

In addition to the percent substitutions discussed previously, there are 26 special variables that you can use to hold information. These variables are named using a `%v` followed by a letter of the alphabet: `%va`, `%vb`, `%vc...%vz`. These variables are known as *registers*.

You can set these registers using `@va`, `@vb`, `@vc...@vz`, along with the object name. Following is the syntax:

```
> @register object = actions
```

Using your rubber duck object created earlier, you can type

```
> @va duck = @emit QUACK!
```

This sets the va register on the duck to the @emit QUACK! action.

To trigger a register, use the trigger command, as in the following:

```
> @trigger object/attribute
```

So, to trigger the va register on the duck, type the following syntax:

```
> @trigger rubber duck/va
QUACK!
```

You then could program the duck to quack whenever someone picks it up or puts it down, as in the following:

```
> @asucc duck = @trigger me/va
Set.
> @adrop duck = @trigger me/va
Set.
> take duck
Taken.
QUACK!
> drop duck
Dropped.
QUACK!
```

TIP

You can program registers to trigger other registers, too.

You also can abbreviate the @trigger command as @tr.

Listening

You can program an object to look for certain words or phrases in the messages that it receives, and then perform special actions when those words or phrases appear. In a way, it is similar to the Secret Word on Groucho Marx's old *You Bet Your Life* television show: a stuffed duck with a hundred-dollar bill would drop down whenever one of the contestants said the "Secret Word."

To listen for a word, use the @listen command.

```
> @listen object = string
```

The *string* is the word or phrase that you are trying to detect. You can use wildcard characters, such as the asterisk (*), to match other parts of a message.

If the string is detected in a message, the @ahear command is executed:

```
> @ahear object = actions
```

You then can program the object to perform special actions when a word is detected.

In the rubber duck example, you could program the rubber duck to say an amusing message whenever someone says the word duck, as in the following:

```
> @listen duck = *duck*
Set.
```

```
> @ahear duck = "Did someone say 'duck'? QUACK!
Set.
> "Have you ever had Peking duck?
Speedy says, "Have you ever had Peking duck?"
rubber duck says, "Did someone say 'duck'? QUACK!"
```

In addition to the `@ahear` attribute, you also can use the `@amhear` and `@aahear` attributes. The `@ahear` attribute is used to detect strings in messages that the listening object did not generate itself, the `@amhear` attribute looks in messages that it generated, and the `@aahear` attribute responds to all messages.

Numbered Variables

When a message is matched by the `@listen` command, portions of the message are stored in special numbered variables, known as *positional parameters*. You denote these variables using `%0`, `%1`, `%2`, `%3`, and so on, up to `%9`. These variables are known collectively as the *stack*.

These numbered variables are set in two ways. One way is through pattern-matching in the `@listen` command. The other way is by directly setting them using the `@trigger` command.

Examine the `@listen` method first. Each numbered variable is set to a wildcard portion of the message that the `@listen` command detects. So, if you use wildcard characters in your `@listen` string, you can access the wildcard portion of the string using these numbered variables.

Each wildcard asterisk matches one word in the triggering message. However, the last asterisk in a string of asterisks matches the remaining wildcard text of the message up to the next trigger word.

You could use these numbered variables to program the rubber duck to detect when someone is talking about it:

```
> @listen duck = * says*rubber duck *
Set.
> @ahear duck = "Are you talking about me, %0?
Set.
```

In the preceding example, the first asterisk in the `@listen` command would match whatever comes before the word `says`, and this text would be stored in `%0`. The second asterisk matches whatever comes between `says` and `rubber duck`, including punctuation. This text is stored in `%1`. The text after `rubber duck` is matched by the third asterisk and stored in `%2`.

You then use the `%0` variable to retrieve the name of the object speaking the text. It is used in the `@ahear` attribute. You have programmed a rather paranoid duck.

```
> "Look at the rubber duck.
Speedy says "Look at the rubber duck."
rubber duck says "Are you talking about me, Speedy?"
Floyd says "This rubber duck is paranoid."
rubber duck says "Are you talking about me, Floyd?"
```

The second way to set the positional parameters is to explicitly set them using the `@trigger` command:

```
> @trigger object/attribute = item1, item2, item3 ... itemN
```

So, if you have programmed the `@va` attribute to be

```
> @va duck = "Are you %1 or %2, %0?
Set.
```

and then you triggered it as follows:

```
> @trigger duck/va = Speedy, sleepy, wide-awake
```

you would see

```
rubber duck says "Are you sleepy or wide-awake, Speedy?"
```

Or, you could enter

```
> @trigger duck/va = Floyd, a sweet little angel, a pesky little devil
rubber duck says "Are you a sweet little angel or a pesky little devil, Floyd?"
```

One small caveat—because commas are used to separate items in the triggering command from one another, you have to put braces around your text if you actually want a comma in your string. Otherwise, your comma will be used as a separator.

The following is code without braces:

```
> @trigger duck/va = Speedy, tall, dark and handsome, red, white and blue
rubber duck says "Are you tall or dark and handsome, Speedy?"
```

The following is code with braces:

```
> @trigger duck/va = Speedy, { tall, dark and handsome }, { red, white and blue }
rubber duck says "Are you tall, dark and handsome or red, white and blue, Speedy?"
```

Just remember that you sometimes may need to enclose your strings in braces to make things work right. This especially is true when programming more complex objects and lengthy code.

User-Defined Commands

Sometimes you want to be able to type a command that is not defined. On MUSHes, you can create your own commands by defining them on an object. Then, when someone enters that command in the room where the object resides, the command will be executed. The following is syntax for defining your own command on an object:

```
> @attribute object = $command:actions
```

You could create a simple quack command on the rubber duck by typing

```
> @va duck = $quack:@emit QUACK!
Set.
> quack
QUACK!
```

When you create your own commands, be sure to name them something different from common commands, such as `page` or `whisper` because the common commands take precedence over user-defined commands.

User-Defined Attributes

You also can define your own attributes on an object. These attributes are useful as named variables in your MUSH code. You can use

```
> &attribute object = anything
```

or

```
> @set object = attribute:anything
```

User-defined attributes frequently are used to store temporary values. For example, you could define an attribute on the duck to store the name of the player who last quacked the duck:

```
> @va duck = $quack:@emit QUACK!;&prevquack me = %n
Set.
> @vb duck = $lastquack:@emit [get(me/prevquack)] quacked me earlier!
Set.
> quack
QUACK!
> lastquack
Speedy quacked me earlier!
```

The `va` register holds the code for the `quack` command. It has two parts. The first part prints out the `QUACK!` message, and the second part stores the name of the player who quacked the duck in the `prevquack` attribute. This `prevquack` attribute is a user-defined attribute.

The `vb` register then retrieves the `prevquack` attribute and uses it in its own message. The `[get(me/prevquack)]` portion is a function, which is discussed shortly. It simply accesses (`get`s) the `prevquack` attribute on the duck (`me`).

Functions

MUSHes have many built-in *functions*. You can use these functions in your own programming code. The following is the syntax for a function call:

```
[ function(parameters...)]
```

The function itself is enclosed in square brackets, and the parameters to the function are enclosed in parentheses and separated by spaces.

Functions can be nested in one another. Nested functions only need one outermost set of square brackets.

Table 15.7 lists some important functions that are used frequently in MUSH code.

Table 15.7. Common MUSH functions.

Function	*Example*	*Meaning*
`get(`*`object/attribute`*`)`	`[get(me/va)]`	Obtains the value of an attribute on the given object.
`eq(`*`item1, item2`*`)`	`[eq(me,#444)]`	Tests to see whether items are equal.
`loc(`*`object`*`)`	`[loc(me)]`	Obtains the object number of an object's location.
`name(`*`object`*`)`	`[name(me)]`	Obtains the name of the given object.
`rand(`*`number`*`)`	`[rand(5)]`	Generates a random number from `0` to `(number-1)`.
`s(`*`string`*`)`	`[s(%P left.)]`	Performs pronoun substitution on the string.
`v(`*`v-register`*`)`	`[v(va)]`	Obtains the value of the given `v`-register.
`v(`*`user-attribute`*`)`	`[v(prevquack)]`	Obtains the value of the given `v`-register.

The following is an example of how you can use these functions:

```
> @emit [name(me)] is in the [name(loc(me))] room.
Speedy is in the Swimming Pool—Deep Side room.
```

The @switch Command

You use the `@switch` command to program `if-then-else` and `case`-statement style expressions in MUSH code. This enables you to test for certain conditions, and then perform actions based on those conditions.

The following is the format of the `switch` command:

```
> @switch test-expression = match1, actions1 , match2, actions2, ..., matchN,
actionsN, default-actions
```

Using this syntax, the *`test-expression`* is evaluated. It is matched against *`match1`*. If these are equal, then the *`actions1`* actions are executed. Otherwise, it tries to match up with *`match2`*. If this matches, then *`actions2`* are executed, and so on. It does this for all the matches in the statement. At the end, if nothing matches, the *`default-actions`*, if any, are executed.

So, you could program the duck to respond differently to male and female players. You can use the `match()` function to see if the sex attribute of the player petting the duck is female, as in the following:

```
> @vf duck = $pet:@switch match(get(%#/sex),female)=1, {@emit The duck wags its
little tail and bats its eyes romantically.}, 0, {@emit The duck smiles and looks
very friendly.}
Set.
```

If someone who is female pets the duck, the duck responds with

```
The duck wags its little tail and bats its eyes romantically.
```

and if a male player pets the duck, then you see the following:

```
The duck smiles and looks very friendly.
```

You also could use the `rand()` function to generate a random number, and use that to determine which message to display, as in the following:

```
> @vf duck = $pet:@switch rand(4)=0,{@emit Hey! Watch where yer pettin!},1,{@emit A
bit lower, please.},2,{@emit You have nice hands.},3,{@emit Brrr! Your hands are
cold!}
Set.
> pet
A bit lower, please.
> pet
Hey! Watch where yer pettin!
> pet
You have nice hands.
> pet
A bit lower, please.
> pet
A bit lower, please.
> pet
Brrr! Your hands are cold!
```

Lists

There are different methods to generate lists of objects on various MUDs, and the functions differ from MUD to MUD. For purposes of discussion here, assume that the `lcon()` function is available on your system. If it isn't, you may have to search through your MUD's help files to find an equivalent function.

The `lcon(`*`object`*`)` function returns a space-separated list of the objects in a room or in a container. The list is in object number format.

If you are in the deep side of the pool

```
> say lcon(here)
You say "#1111 #7500 #3000 "
```

where Speedy is object `#1111`, the floating raft is object `#7500` and the rubber duck is object `#3000`.

Once you have a list of objects, if you want to perform the same command on each item in the list, you can use the `@dolist` command.

```
> @dolist list = actions
```

The list in the preceding command is a space-separated list of items. The command works in conjunction with a special indicator in the actions portion of the command—the `##` indicator. The actions are executed once for each item in the list, and the `##` indicator in the actions is replaced by each item in the list.

So, if you type

```
> @dolist 1 2 3 4= say Number ##
You say "Number 1"
You say "Number 2"
You say "Number 3"
You say "Number 4"
```

Or, you could use list commands in the list portion of the command, as in the following:

```
> @dolist lcon(here) = say This is the [name(##)].
You say "This is the Speedy."
You say "This is the floating raft."
You say "This is the rubber duck."
```

You could use list construction commands and the `@dolist` command to send messages to objects inside of containers. If there was a box with Alfred, Robin, and Selina in it, you could page them using:

```
> @dolist lcon(box) = page ## =Hi there!
You paged Alfred with 'Hi there!'.
You paged Robin with 'Hi there!'.
You paged Selina with 'Hi there!'.
```

Needless to say, the `@dolist` command can be very useful in your MUSH programming.

The Command Queue

Whenever an action occurs, it actually goes into a *queue*. This queue is then processed, action by action. Most of the time, these actions occur immediately; however, you can set up delayed actions, also.

To see your queue, use the `@ps` command. It lists all the commands that are waiting to be executed. If you are programming something using programming language constructs such as `@wait`, you can check the queue to see that your commands have been entered there and are waiting to be executed.

The *@wait* and *@halt* Commands

The `@wait` command creates a delay. Following is the syntax:

```
> @wait seconds = actions
```

The MUD waits the given number of *seconds* before running the specified *actions*.

So, if you type

```
> @wait 10 = "Hey!
```

then 10 seconds would pass before your `Hey!` message would appear.

You can use the `@wait` command to program the floating raft to float around the pool. Every minute, it will wander from the deep side to the shallow side. After another minute, it will wander back. You can use the `@wait` command to set up registers to accomplish this. You then can program two commands, `startfloat` and `stopfloat`, to start the raft on its journey. You can lock the raft so that it cannot be taken, and lock the exits so the raft doesn't leave the pool.

Create the raft on the deep side of the pool.

```
> @va raft = @wait 60 = { @emit The raft floats along.; west; @tr me/vb }
Set.
> @vb raft = @wait 60 = { @emit The raft floats along.; east; @tr me/va }
Set.
> @vc raft = $startfloat:@tr me/va
Set.
> @vd raft = $stopfloat:@halt me
Set.
> @lock raft = me & ! me
Locked.
```

Next, go to the shallow side of the pool and lock the west exit so that the raft cannot go through it. Recall that the raft was object #7500:

```
> @tel me = #1200
Swimming Pool—Shallow Side (#1200R)
You are floating in the shallow side of the pool. The pool gets deeper to the east.
You can see the blue and white tiles on the bottom of the pool clearly.
Obvious exits:
west east
> lock west = ! #7500
Locked.
> east
Swimming Pool—Deep Side (#1201R)
You are bobbing up and down in the deep side of the pool. The pool gets shallow to
the west. A diving platform rises above the water, to the east. You can see the blue
and white tiles on the bottom of the pool faintly through the shimmering water below
you.
Contents:
rubber duck (#3000)
floating raft (#7500)
Obvious exits:
west east down
> lock east = ! #7500
```

Then start the raft on its journey by typing the following:

```
> startfloat
```

After a minute, you will see

```
The raft floats along.
floating raft swims to the shallow side of the pool.
```

And, a minute later, you see the following:

```
floating raft swims over to this side of the pool.
```

If you jump onto the raft by entering it (remember that it is a container object), you will be moved from one side of the pool to the other along with the raft as it moves.

The raft will continue floating from one side of the pool to the other until you run the `stopfloat` command. The `stopfloat` command executes a `@halt` on the object. The syntax of the `@halt` command is:

```
> @halt object
```

This command clears out the queue for that object. Any actions in the queue pertaining to that object are removed.

In addition, the command

```
> @halt
```

clears out your personal queue.

Semaphores

A *semaphore* is a slightly more advanced MUSH feature that controls when actions can occur. Like real-life railroad semaphores that signal when it is safe for a train to take a particular rail line, MUSH semaphores indicate when it is safe for actions to execute.

You might use semaphores to make sure that a series of actions are not interrupted by another series of actions. For example, if you have a machine that creates a magic wand, you may want to make sure someone else does not activate the machine and try to create a wand while the first wand is being handled. You can use semaphores for this.

Semaphores have a count, which is the number of actions that are currently blocked and are waiting to be unblocked. When the count is `0`, actions occur immediately. A number higher than `0` indicates how many actions are currently blocked. A number lower than zero indicates that that many actions will execute immediately.

When you use the `@wait` command with an object, the actions are placed on the object's queue and postponed until a notification occurs. Following is the syntax of the `@wait` command on an object:

```
> @wait object = actions
```

Or, if you want to specify a maximum waiting time, use the following:

```
> @wait object/seconds = actions
```

in which case the *`actions`* will be performed if the waiting time, given in *`seconds`*, expires. This is useful when you want to make sure the actions eventually will be executed.

You can indicate that your queued actions should be executed by using the `@notify` command

```
> @notify object
```

which forces the first action in the enqueued action list to be performed. You also can specify a certain number of actions to be executed using the following:

```
> @notify object = number-of-actions
```

The `@drain` command resets the semaphore count on an object to `0`.

The `@startup` command specifies actions that will be executed when the MUSH starts up.

As an example, you can create a guppy generator in the shallow side of the pool.

```
> @create guppy generator
guppy generator created as object #888.
> @desc guppy = This strange contraption creates guppies. Type 'guppy' to make some
guppies!
Set.
> @startup guppy = @drain me; @notify me
Set.
> @notify generator
Set.
```

Now the guppy generator has a count of `-1`. This means that any actions on it will execute immediately. It also is set up to notify itself when the MUSH starts up.

You now can program the `guppy` command.

```
> @va guppy = $guppy:@wait me = {@create a guppy; @desc a guppy=You see a small
fish.;@set a guppy=DESTROY_OK;@tel a guppy=[loc(%#)];@emit You made a guppy!;@notify
me}
```

The `guppy` command creates a guppy, sets its description and some flags, and then teleports the guppy into the same room as the enactor of the command.

```
> drop guppy
Dropped.
> guppy
You made a guppy!
```

If you look at the room contents now, you will see that a new guppy object has been created. If you continue to type `guppy`, more and more guppies will appear.

If you really wanted the guppies to be proliferous, you could program the guppies to respond to the `guppy` command also, and create their own guppies. If you do this, however, please remember to `@destroy` all your guppies when you are finished with your example. There's no sense in having all those guppies taking up space in the MUD database. But it is kind of amusing to experiment!

Puppets

In a MUSH, there are special objects called *puppets*. These objects are similar to puppets in real life because their owner can control them. Puppets can see and hear things in a room, and they can report what they see and hear back to their owners.

To change an object into a `puppet`, type the following:

```
> @set object = puppet
```

The `puppet` object will announce that it has grown ears and can now hear.

To force an object to perform an action, use the `@force` command, as in the following:

```
> @force object = actions
```

You can use the `@force` command along with puppets to control what they do.

Assume that you are in the Poolside East room, under the diving board.

```
> @create Lifeguard
Lifeguard created as object #986
> @set Lifeguard=puppet
Lifeguard grows ears and can now hear.
Lifeguard> Lifeguard grows ears and can now hear.
Flag set.
> inv
You are carrying:
Lifeguard(#986p)
You have 595 pennies.
> drop Lifeguard
Lifeguard> Speedy dropped you.
Lifeguard has left.
Lifeguard has arrived.
Lifeguard> Poolside East(#1207R)
Lifeguard> You are standing on the east side of the pool, near the deep end. The pool
is to your west. A diving platform rises above you here, reachable by a ladder.
Lifeguard> Contents:
Lifeguard> Speedy(#1111Pc)
Lifeguard> Obvious exits:
Lifeguard> up west
Dropped.
> @force Lifeguard = "Hey, you kids, quit horsing around in the pool!
Lifeguard> You say "Hey, you kids, quit horsing around in the pool!"
Lifeguard says "Hey, you kids, quit horsing around in the pool!"
> @force Lifeguard = west
Lifeguard has left.
Lifeguard> Swimming Pool – Deep Side(#1201R)
Lifeguard> You are bobbing up and down in the deep side of the pool. The pool gets
shallow to the west. A diving platform rises above the water, to the east. You can
see the blue and white tiles on the bottom of the pool faintly through the shimmering
water below you.
Lifeguard> Contents:
Lifeguard> floating raft(#7500e)
Lifeguard> rubber duck(#3000)
Lifeguard> Obvious exits:
Lifeguard> east down west
> @force #986=enter raft
Lifeguard> You jump onto the floating raft!
Lifeguard> floating raft(#7500e)
Lifeguard> You are on top of an inflated, floating raft in the pool.
> @force #986="I'm on the raft!
Lifeguard> You say "I'm on the raft!"
Lifeguard says "I'm on the raft!"
```

@Command Reference

Following is a list of the @commands that are available on MUCKs and MUSHes. Your particular MUD may not implement all of these, however. Some commands are only available on MUSHes, and are denoted as such.

`@@`	Does nothing. Useful for comments in your code.
`@aahear` *`object=actions`*	Sets the *`actions`* triggered when the `@listen` string is matched by a pose or utterance from the *`object`* itself or from a player. (MUSH only)
`@aclone` *`object=actions`*	Sets the *`actions`* triggered by the newly cloned copy when the *`object`* is cloned. (MUSH only)
`@aconnect` *`object=actions`*	Sets the *`actions`* triggered when someone connects onto the MUD. (MUSH only)
`@adescribe` *`object=actions`*	Sets the *`actions`* triggered when the *`object`* is looked at. (MUSH only)
`@adfail` *`object=actions`*	Sets the *`actions`* triggered when someone fails to drop the *`object`* due to a drop lock. (MUSH only)
`@adisconnect` *`object=actions`*	Sets the *`actions`* triggered when someone disconnects from the MUD. (MUSH only)
`@adrop` *`object=actions`*	Sets the *`actions`* triggered when the *`object`* is dropped. (MUSH only)
`@aefail` *`room¦object¦player=actions`*	Sets the *`actions`* triggered when someone fails to enter the *`room`*, *`object`*, or *`player`*. (MUSH only)
`@aenter` *`room¦object¦player=actions`*	Sets the *`actions`* triggered when the *`room`*, *`object`*, or *`player`* is entered. (MUSH only)
`@afail` *`object=actions`*	Sets the *`actions`* triggered when an attempt to use the *`object`* fails. (MUSH only)
`@agfail` *`object=actions`*	Sets the *`actions`* triggered when someone fails to give away the *`object`* due to a give lock. (MUSH only)
`@ahear` *`object=actions`*	Sets the *`actions`* triggered when the *`object`*'s `@listen` string is matched. (MUSH only)

`@akill` *`object=actions`*	Sets the *`actions`* triggered when the *`object`* is killed, after it has returned to its home. (MUSH only)
`@aleave` *`room¦object=actions`*	Sets the *`actions`* triggered when a player or other object leaves the specified *`room`* or *`object`*. (MUSH only)
`@alfail` *`room¦object=actions`*	Sets the *`actions`* triggered when someone fails an attempt to leave the *`room`* or *`object`*. (MUSH only)
`@alias` *`player=alias`*	Sets the *`alias`* by which to reference the *`player`*. (MUSH only)
`@amhear` *`object=actions`*	Sets the *`actions`* triggered when the object's `@listen` string is matched by a message from the *`object`* itself. (MUSH only)
`@amove` *`object=message`*	Sets the *`message`* emitted if the *`object`* moves by any means. (MUSH only)
`@apay` *`object=actions`*	Sets the *`actions`* triggered when someone or something gives credits to the *`object`*. (MUSH only)
`@arfail` *`object=actions`*	Sets the *`actions`* triggered when the *`object`* fails to receive an object that was given to it due to a give lock. (MUSH only)
`@asuccess` *`object=actions`*	Sets the *`actions`* triggered when the *`object`* is successfully used. (MUSH only)
`@atfail` *`object=actions`*	Sets the *`actions`* triggered by the *`object`* when someone tries to teleport to the *`object`*, but fails. (MUSH only)
`@atport` *`object=actions`*	Sets the *`actions`* triggered when the *`object`* teleports somewhere. The *`actions`* occur after the *`object`* has teleported. (MUSH only)
`@aufail` *`object=actions`*	Sets the *`actions`* triggered when someone fails to use the *`object`* due to a use lock. (MUSH only)
`@ause` *`object=actions`*	Sets the *`actions`* triggered when someone uses the *`object`* via the `use` command. (MUSH only)

`@away player=message`	Sets a *message* on the *player* that is displayed to someone who tries to page the *player* when the *player* is not connected to the MUD. (MUSH only)
`@charges object=#charges`	Limits the number of times an *object* can be used. (MUSH only)
`@chown object=player`	Changes the ownership of the *object* to the specified *player*. You must own the *object* or the *object* must be `CHOWN_OK`.
`@clone object`	Creates a duplicate of the *object*. (MUSH only)
`@cost object=#credits`	Sets the number of *credits* that must be given to the *object* to trigger `@pay`, `@opay`, and `@apay`. (MUSH only)
`@create obj-name[=#credits]`	Creates a new object whose name is the specified *obj-name*. The command costs *#credits* or 10 credits, whichever is greater. The object's "actual" value will become `(#credits / 5) - 1`.
`@decompile`	Outputs a series of commands that can be fed back into the MUD to redefine and reset all of an object's registers and flags. This command can be used to save an object to disk, edit it, and read it back in. (MUSH only)
`@describe object=text`	Sets the *text* seen when the *object* is looked at.
`@destroy object`	Sets the *object* so as to be destroyable and returns the creator's investment. (MUSH only)
`@dfail object=message`	Sets the *message* seen when someone fails to drop the *object* due to a drop lock. (MUSH only)
`@dig room-name[=exits[,return-exits]]`	Creates a room, giving it the specified *room-name*. The optional portions allow alternate names and auto-linked exits on MUSHes.
`@doing message`	Sets the player's `@doing` *message*, which is displayed in the `WHO` list. (MUSH only)

`@dolist` *list=actions*	Performs the *actions* on each of the items in the *list*, one by one, by substituting the special symbol ## in the *actions* by each item in the *list*. (MUSH only)
`@drain` *object*	Clears out the semaphore *object* and resets it to its initial state. (MUSH only)
`@drop` *object=message*	Sets the *message* seen when someone drops the *object*. If entered without a *message*, it clears any existing message.
`@ealias` *object=alias-list*	Sets a list of aliases that can be used to enter the *object*, instead of "`enter` *object*".
`@edit` *MUF-program*	Searches for the given *MUF-program* and, if found, puts the user into edit mode. (MUCK only)
`@edit` *object/attribute={old-string},{new-string}*	Replaces the first occurrence of the *old-string* by the *new-string*. The *attribute* can be any attribute on the *object* that holds a string value (such as `@desc`, `@succ`, `@adrop`, `@listen`, `@ahear`, `@va`, `@vb`, and so on). If the strings contain only alphabetic characters, the curly braces may be omitted. (MUSH only)
`@efail` *object=message*	Sets the *message* shown to a player who fails to enter the *object*. (MUSH only)
`@emit` *message*	Emits the *message* to everything in the room. (MUSH only)
`@enter` *object=message*	Sets the *message* seen when a player enters the *object*. (MUSH only)
`@entrances` *room¦object*	Lists everything linked to the *room* or *object*. You must control the *room* or *object*.
`@fail` *object=message*	Sets the *message* seen when an attempt to use the *object* fails.
`@femit` *object=message*	Forces the *object* to emit the *message*. You must own the *object*. (MUSH only)
`@filter` *object=pattern1 [, ..., patternN]*	Sets the *patterns* to be looked for to suppress text on the *object* generated because of the AUDIBLE flag. (MUSH only)

`@find name`	Displays the name and number of every object you control whose name matches the specified *name*. Usually costs 100 credits on MUSHes, but may be higher on some MUSHes.
`@force player¦object=actions`	Forces the specified *player* or *object* to perform the *actions*, as though the *player* or *object* entered the *actions* itself.
`@forwardlist object=database-reference-list`	Sets a list of locations that will receive messages heard by an *object* that has its AUDIBLE flag set. (MUSH only)
`@fpose object=message`	Forces the *object* to pose the *message*. You must own the *object*. (MUSH only)
`@gfail object=message`	Sets the *message* seen by a player when the *object* fails to be given away by the player due to a give lock. (MUSH only)
`@halt [object]`	Stops a process or a runaway machine. (MUSH only)
`@idescribe object=message`	Sets the *message* seen when the *object* is entered or looked at from the inside. (MUSH only)
`@idle player=message`	Sets the *message* sent to people when they page you, used to indicate that you are idle. (MUSH only)
`@infilter object=pattern1 [, ..., patternN]`	Sets the *patterns* to be looked for to suppress text sent to the contents of *object* generated due to `@listen`. (MUSH only)
`@inprefix object=prefix`	Sets a *prefix* to be prepended to text sent to the contents of the *object* by `@listen`. (MUSH only)
`@kill object=message`	Sets the *message* seen by someone who `kills` *object*. (MUSH only)
`@lalias object=alias-list`	Sets a list of aliases that can be used to leave the *object*, instead of "`leave` *object*". (MUSH only)
`@last player`	Displays a short history of connection attempts for the *player*. You can only obtain history about yourself.
`@leave object=message`	Sets the *message* shown to a player upon leaving the *object*. (MUSH only)

`@lfail` *`object=message`*	Sets the *`message`* shown to a player who fails to leave the *`object`*. (MUSH only)
`@link` *`object=room`*	For things and players, makes the specified *`room`* `Home`. For rooms, makes the specified *`room`* the `drop-to room`. For exits, makes the specified *`room`* the exit's target room.
`@list` *`MUF-program`*	Lists out the given *`MUF-program`*. (MUCK only)
`@list` *`option`*	Lists information about the database, based on the value of the given *`option`*. This *`option`* can be `attributes`, `commands`, `costs`, `default_flags`, `flags`, `functions`, `options`, `switches`. Leaving off the *`option`* will give you a list of possible options. (MUSH only)
`@listen` *`object=string`*	Listens for the given *`string`* to trigger `@ahear`, `@amhear`, and/or `@aahear`. If the *`string`* is matched, the *`object`*'s contents also hear the message. (MUSH only)
`@listmotd`	Displays the current message-of-the-day. (MUSH only)
`@lock` *`object=lock`*	Sets the *`lock`* on the *`object`*. Only players or things satisfying the lock will be able to "succeed" with the object (pick up a thing, go through an exit, and so on). (MUSH only)
`@move` *`object=message`*	Sets the *`message`* shown to the *`object`* itself when it moves by any means. (MUSH only)
`@mvattr` *`object=old,new[,copy]`*	Renames the attribute named *`old`* on the *`object`* to the attribute *`new`*, and is copied to *`copy`* if *`copy`* is specified. (MUSH only)
`@name` *`object=new-name`*	Changes the *`object`*'s name.
`@notify` *`object[=count]`*	Notifies the semaphore *`object`*, running *`count`* commands that have been waiting on the *`object`*. If less than *`count`* commands are waiting, then the semaphore will be set so that *`count`* later `@wait` commands will immediately execute. (MUSH only)

`@odescribe` *`object=message`*	Sets the *`message`* that is seen by the other players in the room when a player looks at the specified *`object`*. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@odfail` *`object=message`*	Sets the *`message`* seen by other players in the room when someone fails to drop the *`object`* due to a drop lock. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@odrop` *`object=message`*	Sets the *`message`* that is seen by the other players in the room when a player drops the specified *`object`*. The *`message`* is prefaced by the name of the triggering player. If the *`object`* is a room, this sets the *`message`* that is displayed to other people when someone enters the room.
`@oefail` *`object=message`*	Sets the *`message`* shown to the other players in the room if a player fails to enter the specified *`object`*. (MUSH only)
`@oemit` *`object=message`*	Sets the *`message`* to be emitted to everything in the room except the *`object`* itself. (MUSH only)
`@oenter` *`object=message`*	Sets the *`message`* that is seen by the other players in a thing when a player enters that thing. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@ofail` *`object=message`*	Sets the *`message`* that is seen by the other players in the room when a player fails an attempt to use the specified *`object`*. The *`message`* is prefaced by the name of the triggering player.
`@ogfail` *`object=message`*	Sets the *`message`* seen by other players in the room when someone fails to give the *`object`* away due to a give lock. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@okill` *`object=message`*	Sets the *`message`* seen by other players in the room when the *`object`* is killed. The *`message`* is prefaced by the name of the object being killed. (MUSH only)

`@oleave` *`object=message`*	Sets the *`message`* that is seen by the other players in a thing when a player leaves that thing. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@olfail` *`object=message`*	Sets the *`message`* that is seen by the other players in a thing when a player fails to leave the specified thing. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@omove` *`object=message`*	Sets the *`message`* seen by all other objects in the room the *`object`* moves to when the specified *`object`* moves by any means. (MUSH only)
`@opay` *`object=message`*	Sets the *`message`* that is seen by the other players in the room when a player or thing pays the specified *`object`*. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@open` *`direction`*`[;`*`other-directions`*`] [=#`*`room`*`]`	Creates an exit in the specified *`direction`*(s). If a *`room`* number is specified, the exit is linked to that *`room`*. Otherwise, the exit remains unlinked. Anyone can use `@link` to specify where an unlinked exit leads.
`@orfail` *`object=message`*	Sets the *`message`* seen by other players in the room when someone fails to receive the *`object`* due to a receive lock. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@osuccess` *`object=message`*	Sets the *`message`* that is seen by the other players in the room when a player or thing succeeds in using the specified *`object`*. The *`message`* is prefaced by the name of the triggering player.
`@otfail` *`object=message`*	Sets the *`message`* seen by other players in the room when someone fails to teleport somewhere. The *`message`* is prefaced by the name of the triggering player. (MUSH only)

`@otport` *`object=message`*	Sets the *`message`* seen by other players in the room when the *`object`* teleports into their room. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@oufail` *`object=message`*	Sets the *`message`* seen by other players in the room when someone fails to use the *`object`* due to a use lock. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@ouse` *`object=message`*	Sets the *`message`* seen by other players in the room when someone succeeds in using the *`object`*. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@owned`	Lists the objects you own. (MUCK only)
`@oxenter` *`object=message`*	Sets the *`message`* seen by other players in the room being left when someone enters the *`object`*. The *`message`* is shown to those outside the *`object`*. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@oxleave` *`object=message`*	Sets the *`message`* seen by other players in the room being entered when someone leaves the *`object`*. The *`message`* is shown to those outside the *`object`*. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@oxtport` *`object=message`*	Sets the *`message`* seen by other players in the room when the *`object`* teleports out of their room. The *`message`* is prefaced by the name of the triggering player. (MUSH only)
`@parent` *`object[=parent]`*	Sets the parent of the *`object`* to the given *`parent`*, or clears the parent if *`parent`* is omitted. You must control *`object`* and own *`parent`*. (MUSH only)
`@password` *`old-password=new-password`*	Changes your password.
`@pay` *`object=message`*	Sets the *`message`* shown to the player who pays the specified *`object`*. (MUSH only)

`@pemit` *`player=message`*	Emits the *`message`* to a specified *`player`*. (MUSH only)
`@prefix` *`object=prefix`*	Sets a *`prefix`* that will be prepended to all messages sent out by the *`object`* due to the `AUDIBLE` flag. (MUSH only)
`@ps`	Lists the queued commands that have not yet been executed. (MUSH only)
`@quota`	Displays your current builders' quota. (MUSH only)
`@recycle` *`object`*	Recycles the *`object`*, which saves space in the database. (MUCK only)
`@reject` *`player=message`*	Sets the *`message`* sent to all players who fail to page you due to your page lock. (MUSH only)
`@rfail` *`object=message`*	Sets the *`message`* displayed to the player who fails to give the *`object`* due to a receiver failing to pass the *`object`*'s receive lock. (MUSH only)
`@robot` *`name=password`*	Creates a robot named *`name`*, and owned by you. Costs 1000 credits. (MUSH only)
`@runout` *`object=actions`*	Sets the *`actions`* triggered when the *`object`*'s charges reach 0. (MUSH only)
`@search` *`[player] [class=restrictions]`*	Searches the MUD database and lists the objects that meet the player's search criteria. If the *`player`* argument is supplied, only objects owned by that player will be listed. If the *`class`* argument is supplied, only objects of a certain class will be listed. Costs 100 credits. The command `@search flags=RHD` would list all rooms (the `R` flag) set `HAVEN` and `DARK` (the `H` and `D` flags). The command `@search type=rooms` would list all rooms owned by the player. (MUSH only)
`@set`	Used to set attributes and flags on objects. May be used in various ways, see the following specific `@set` descriptions.

`@set object=flag`	Sets the specified *flag* on the *object*.
`@set object=! flag`	Resets the specified *flag* on the *object*.
`@set object=attribute:value`	Sets an *attribute* value on the *object*.
`@attribute object=value`	Short form of `@set object attribute:value`. User-defined attributes cannot be set in this manner.
`@sex player=gender`	Sets the *gender* of the *player*. The player's gender is used for pronoun substitution. (Options: `male`, `female`, `neuter`, `plural`. The default is neuter.) Pronouns cannot be used in reference to players unless their gender is set to one of these choices. Visible to all. (MUSH only). On MUCKs, the command is `@set me=sex:gender`.
`@startup object=actions`	Sets the *actions* to be performed by the *object* in the event that the MUD gets rebooted. By using `@startup`, you can retrigger objects that need to be running continuously. (MUSH only)
`@stats [player]`	Displays statistics about all the objects in the game, or about the given *player* if supplied.
`@success object[=message]`	Sets the success *message* for the specified *object*, which is displayed to the player whenever a player successfully uses the object.
`@sweep`	Lists all the objects and players that are listening in the room you currently are in, as well as in the objects you are carrying. In these listings, `player` denotes a connected player that hears all occurrences in the room, `puppet` denotes a puppet belonging to a connected player, relaying all occurrences in the room to the owner, `messages` denotes an object that is listening for specific occurrences in the room, and `commands` denotes an object waiting for a specific command. (MUSH only)

`@switch`	Evaluates a conditional expression and performs different actions based on the results of that evaluation. (MUSH only) May be used in two ways:
`@switch` *`condition={pattern1},{yes-actions},{no-actions}`*	Corresponds to the `if-then-else` programming structure.
`@switch` *`condition={pattern1},{yes-actions1},{pattern2},{yes-actions2},...,{default-actions}`*	Corresponds to the `case` or `switch` programming structure.
`@teleport [`*`thing`*`=]#`*`room`*	Teleports the given *`thing`* to the specified *`room`*. If the *`thing`* is omitted, the command teleports you to the specified room. You must own or control the *`thing`* or its current location. You can only teleport objects into rooms or objects you own or that are set `JUMP_OK`. If the target room has a `drop-to`, the object will go to the `drop-to` room instead.
`@tfail` *`object=message`*	Sets the *`message`* displayed to a player who fails to teleport to the *`object`*. (MUSH only)
`@tport` *`object=message`*	Sets the *`message`* displayed to an *`object`* whenever it teleports to another location. (MUSH only)
`@trigger`	Passes control and data (on the stack) among items. If you create attributes that are triggered by other commands or actions, you can use this command to trigger them. Many tricky things can be done with simple command combinations. (MUSH only)
`@ufail` *`object=message`*	Sets the *`message`* displayed to someone who fails to use the *`object`* due to a use lock. (MUSH only)
`@use` *`object=message`*	Sets the *`message`* displayed to the player who successfully uses the *`object`*. (MUSH only)

`@unlink` *exit* or `@unlink here`	Removes links from exits. The first version shown removes a link from the specified *exit*. The second removes the `drop-to` on the room. Be careful, however; anyone can relink an unlinked exit, thereby becoming its new owner.
`@unlock` *object*	Removes the lock on the *object*.
`@va` *object=actions*	Sets the v-register on the *object* to the
`@vb` *object=actions*	specified *actions*. Every object has 26
.	built-in registers, `va` through `vz`.
.	Anything may be stored on a v-register.
.	(MUSH only)
`@vz` *object=actions*	
`@version`	Displays the MUCK version number. (MUCK only)
`@wait` *#seconds=actions*	Queues *actions*. The *actions* are placed on the queue and will be executed no earlier than *#seconds* from the time they are queued. The *actions* may be a list of commands in curly braces. (MUSH only)
`@wipe` *object*`[`*/attribute-pattern*`]`	Clears all the attributes on the *object*, or the attributes that match the given *attribute-pattern*, if it is specified. (MUSH only)

MUSH Function Reference

The following is a list of functions that are available on MUSHes. Some functions may not be implemented on your system.

`abs(`*number*`)`	Returns the absolute value of the *number*.
`acos(`*number*`)`	Returns the arccosine of the *number*.
`add(`*a*, *b*`)`	Returns the sum of *a* and *b*.
`after(`*string1*, *string2*`)`	Returns the part of *string1* that occurs after *string2*. Returns a null string if *string2* does not occur in *string1*.

`and(boolean1, boolean2, ..., booleanN)`	Returns 1 if each *`boolean`* argument evaluates to true.
`aposs(object)`	Returns the absolute possessive pronoun (`his`, `hers`, `its`, `theirs`) that corresponds to the *`object`*, based on the `@sex` attribute of the *`object`*.
`asin(number)`	Returns the arcsine of the *`number`*.
`atan(number)`	Returns the arctangent of the *`number`*.
`capstr(string)`	Returns the *`string`*, with its first letter capitalized.
`cat(string1, string2, ... stringN)`	Returns the concatenation of *`string1`* through *`stringN`*. Each string will be separated from one another by a space.
`ceil(number)`	Returns the smallest integer that is greater than or equal to the given *`number`*.
`center(string, width)`	Returns the *`string`*, center-justified, in a field of the given *`width`*.
`comp(n, m)`	Returns `0` if *n* = *m*, `-1` if *n* < *m*, and `1` if *n* > *m*. Alphanumeric strings are compared alphabetically.
`con(object)`	Returns the first thing in the contents list of the *`object`*. Similar to the LISP `car()` function.
`conn(player)`	Returns the number of seconds that the *`player`* has been connected to the MUD, or `-1` if the *`player`* is not connected.
`controls(object1, object2)`	Returns `1` if *`object1`* controls *`object2`*, otherwise returns `0`.
`convsecs(seconds)`	Converts the given *`seconds`* value into a date/time string based on seconds elapsed since January 1, 1970.
`convtime(string)`	Converts the given date/time *`string`* to the number of seconds elapsed since January 1, 1970.
`cos(number)`	Returns the cosine of the given *`number`*.
`delete(string, index, numchars)`	Returns the remainder of the given *`string`* after deleting *`numchars`* characters beginning with the character at the given *`index`*. Characters are numbered starting at `0`.

`dist2d(x1, y1, x2, y2)`	Returns the distance between `(x1, y1)` and `(x2, y2)` as if plotted on a 2-dimensional graph.
`dist3d(x1, y1, z1, x2, y2, z2)`	Returns the distance between `(x1, y1, z1)` and `(x2, y2, z2)` as if plotted on a 3-dimensional graph.
`div(a, b)`	Returns `a` divided by `b` (this is integer division with no remainder).
`e()`	Returns the number `e`.
`edit(string, pattern, replacement)`	Returns the `string` with all occurrences of `pattern` changed to `replacement`. If `pattern` is `"$"`, then `replacement` will be appended to `string`. If `pattern` is `^`, then `replacement` will be prepended to `string`.
`elock(object1[/lock],object2)`	Returns `1` if `object2` would pass the named lock on `object1`.
`eq(integer1, integer2)`	Returns `1` if `integer1` equals `integer2`, `0` otherwise.
`escape(string)`	Parses the `string` and puts an escape character at the start of the `string` and in front of special characters.
`exit(object)`	Returns the first exit in the exit list of the `object`.
`exp(number)`	Returns `e` raised to the power of the given `number` (reverse `ln`).
`extract(string, wordnumber, numwords)`	Returns the substring from `string` starting at word number `wordnumber` and containing `numwords` words. Words are numbered starting at `1`.
`fdiv(a, b)`	Returns `a` / `b`, where `a` and `b` are floating point numbers.
`filter([object/]attribute, list[, delimiter])`	Evaluates the `attribute`, passing each item in `list` as `%0` to the `attribute`. Returns a list of each item that evaluated to `1`, delimited by a space or the given optional `delimiter`.
`first(string)`	Returns the first word of the `string`.
`flags(object)`	Returns a string consisting of the current flags on the `object`.

floor(*number*)	Returns the largest integer less than or equal to the given *number*.
fold([*object*/]*attribute*, *list*[,*base-case*, *delimiter*])	Processes the *list* into the *attribute*, passing the result of each iteration as %0 and the next item in the *list* as %1. The *base-case* is used in the first iteration as %0 if it is provided; otherwise, the first list item is passed as %0 and the second item as %1. You can supply an optional *delimiter* if you want to use a delimiter other than a space.
fullname(#*database-reference*)	Returns the full name of the object with the given *database-reference*.
get(*object*/*attribute*)	Returns the specified *attribute* on the given *object*.
get_eval(*object*/*attribute*)	Returns the specified *attribute* on the given *object*, but first performing %-substitutions and function calls on the *attribute*'s value.
gt(integer1, integer2)	Returns 1 if *integer1* is greater than *integer2*; otherwise, returns 0.
gte(*integer1*, *integer2*)	Returns 1 if *integer1* is greater than or equal to *integer2*, 0 otherwise.
hasflag(*object*, *flag*)	Returns 1 if the *object* has the specified *flag*; otherwise, 0.
home(*object*)	Returns the *object*'s home.
idle(*user*)	Returns the time in seconds that the *user* has been idle. If the *user* is not connected, nonexistent, or hidden from you, returns -1.
index(*list*, *character*, *first-item*, *num-items*)	Returns *num-items* items, starting from the *first-item* in the *list*, using the given *character* to delimit items in the *list*. This allows items in the *list* to be more than one word.
insert(*list*, *position*, *word* [,*delimiter*])	Returns the *list* with the given *word* inserted into the *position*-th position in the list. The *list* begins at element 1. The optional *delimiter* may be used to specify a delimiter other than a space.

`isdbref(string)`	Returns `1` if the `string` is a valid database reference, `0` otherwise.
`isnum(argument)`	Returns `1` if the `argument` is a digit, `0` if the `argument` is a letter or a symbol.
`iter(list, eval-string [, delimiter])`	Evaluates each item in the `list` through the `eval-string`, using the optional `delimiter` to delimit items in the `list`. The special symbol `##` in the `eval-string` will be replaced by each item in the `list`.
`lattr(object)`	Returns a list of the attributes on the given `object` that have a non-null value.
`lcon(object)`	Returns a space-separated list of items from the `object`.
`lcstr(string)`	Converts the `string` to all uppercase letters.
`ldelete(list, position [,delimiter])`	Deletes the item at the `position`-th position from the `list`, using the optional `delimiter` as a delimiter.
`lexits(object)`	Returns a space-separated list of exits on the `object`.
`ljust(string, width)`	Returns the `string`, left-justified, in a field of the given `width`.
`ln(number)`	Returns the natural log of the `number`.
`lnum(number)`	Returns a list of consecutive numbers from `0` to (`number -1`).
`loc(object)`	Returns the database reference of the location of the `object`.
`locate(object, string, flags)`	Uses the given `object` to search for a given `string`, based on the values of the given `flags`. The `flags` are such things as `i` for inventory, `e` for exits, and so on. Consult the MUSH help for more detail.
`lock(object[/lock])`	Returns the named `lock` on the `object`, or the default lock if no `lock` is specified.
`log(number)`	Returns the logarithm base 10 of the `number`.
`lt(integer1, integer2)`	Returns `1` if `integer1` is less than `integer2`; otherwise, returns `0`.

`lte(`*`integer1`*`, `*`integer2`*`)`	Returns `1` if *`integer1`* is less than or equal to *`integer2`*, `0` otherwise.
`lwho()`	Returns a list of object numbers of connected users.
`map([`*`object`*`/]attribute, `*`list`*` [, `*`delimiter`*`])`	Passes each item in *`list`* to the *`attribute`* as `%0`, and forms a new list from the result. Delimits the new list using spaces or the optional *`delimiter`*.
`match(`*`string1`*`, `*`string2`*`)`	Returns `1` if *`string1`* matches *`string2`*; otherwise, returns `0`. *`string2`* may contain wildcards.
`max(`*`number1`*`, `*`number2`*`, ..., `*`numberN`*`)`	Returns the largest number in the list of arguments.
`member(`*`list`*`, `*`item`*`)`	Locates the position of the given *`item`* in the *`list`*, where the given *`list`* is a series of words and/or numbers, separated by spaces. Returns `0` if the *`item`* is not in the *`list`*.
`merge(`*`string1`*`, `*`string2`*`, `*`character`*`)`	Merges *`string1`* and *`string2`* based on the given *`character`*. If a character in *`string1`* is the same as the given *`character`*, then the corresponding character in the same position in *`string2`* is substituted into *`string1`*. The two strings must be of equal length.
`mid(`*`string`*`, `*`index`*`, `*`numchars`*`)`	Returns *`numchars`* characters from the *`string`*, starting at the specified *`index`*.
`min(`*`number1`*`, `*`number2`*`, ..., `*`numberN`*`)`	Returns the smallest number in the list of arguments.
`mod(`*`a`*`, `*`b`*`)`	Returns *`a`* mod *`b`* (the remainder after integer division).
`money(`*`object`*`)`	Returns an integer equal to either the amount of money *`object`* has, if it is a player, or the value of the *`object`* itself.
`MUDname()`	Returns the name of the MUD.
`mul(`*`a`*`, `*`b`*`)`	Returns *`a`* multiplied by *`b`*.

`name(`*`object`*`)`
`name(#`*`database-reference`*`)`
	Returns the name of the *`object`*. The *`object`* itself or a *`database-reference`* can be used as the argument.

`nearby(`*`object1`*`, `*`object2`*`)`	Checks to see if *`object1`* and *`object2`* are near each other; returns `1` if they are, `0` if they aren't. Two objects are considered nearby if they are in the same room, or if one is inside the other one.
`neq(`*`integer1`*`, `*`integer2`*`)`	Returns `1` if *`integer1`* is not equal to *`integer2`*, `0` otherwise.
`next(`*`thing`*`)`	Returns the next non-`DARK` exit in a room, if the *`thing`* is an exit. If the *`thing`* is an object, returns the next item in the inventory list that the object is in. Otherwise returns `#-1`.
`not(`*`boolean`*`)`	Returns the logical `NOT` of the given *`boolean`* value.
`num(`*`object`*`)`	Returns the database reference of the *`object`*.
`obj(`*`object`*`)`	Returns the objective pronoun for the *`object`* (`him`, `her`, `it`, or `them`), based on the `@sex` attribute of the *`object`*.
`or(`*`boolean1`*`, `*`boolean2`*`, ..., `*`booleanN`*`)`	Returns `1` if any *`boolean`* argument evaluates to true, `0` otherwise.
`owner(`*`object`*`)`	Returns the database reference of the *`object`*'s owner.
`parent(`*`object`*`)`	Returns the database reference of the *`object`*'s parent, `#-1` if the *`object`* cannot be found, or if you are not the owner of *`object`* and the object is not set `VISUAL`.
`parse(`*`list`*`, `*`eval-string`*` [,`*`delimiter`*`])`	Evaluates the *`given eval-string`* by taking each item in the *`list`* and substituting it for `##` in the *`eval-string`*, returning a new list of the results, separated by spaces or the optional *`delimiter`*.
`pi()`	Returns pi.
`pos(`*`string1`*`, `*`string2`*`)`	Returns the position in *`string2`* where *`string1`* first appears.

`poss(object)`	Returns the possessive pronoun for the *object* (his, her, its, or their), based on the `@sex` attribute of the *object*.
`power(a, b)`	Returns *a* to the *b* power, where *a* and *b* are floating point numbers.
`r(register-number)`	Returns the value of the given numbered register.
`rand(number)`	Returns a random number between `0` and (*number* -1).
`remove(string, wordnumber, numwords)`	Returns the remainder of the *string* after removing *numwords* words starting at word number *wordnumber*. Words are numbered starting at 1.
`repeat(character, number)`	Returns a string made up of the given *character*, repeated the given *number* of times.
`replace(list, position, word [,delimiter])`	Replaces the *position*-th word in the given *list* with the given *word*, using a space or the optional *delimiter* to delimit words in the list.
`rest(string)`	Returns everything but the first word of *string*.
`reverse(string)`	Reverses the *string*.
`revwords(string, [,delimiter])`	Reverses the order of the words in the *string*, using spaces to delimit words, or the optional *delimiter*.
`rjust(string, width)`	Returns the *string*, right-justified, in a field of the specified *width*.
`rloc(object, levels)`	Returns the location of the *object*'s location, making *levels* nested `loc()` calls at most.
`room(object)`	Returns the database number of the containing room that the *object* is in.
`round(number, places)`	Rounds *number* to `places` decimal places.
`s(string)`	Performs pronoun substitution on the *string*.
`search([player] [class=restriction])`	Returns a list of objects that match the search criteria. Operates the same way as `@search`.

`secs()`	Returns the number of seconds since January 1, 1970.
`secure(`*`string`*`)`	Replaces escaped characters in the *`string`* with spaces, so that the *`string`* will not perform special character escapes.
`setdiff(`*`list1`*`, `*`list2`*`)`	Set difference. Returns a sorted list of the elements in *`list1`* that are not in *`list2`*.
`setinter(`*`list1`*`, `*`list2`*`)`	Set intersection. Returns a sorted list of the elements that are in both *`list1`* and *`list2`*.
`setq(`*`register-number`*`, `*`string`*`)`	Copies the given *`string`* into numbered register *`register-number`*.
`setunion(`*`list1`*`, `*`list2`*`)`	Set union. Returns a sorted list of the elements in *`list1`* and *`list2`*, combined, minus any duplicates.
`sign(`*`number`*`)`	Returns `1`, `0`, or `-1`, depending on whether the *`number`* is positive, zero, or negative.
`sin(`*`number`*`)`	Returns the sine of the given *`number`*.
`sort(`*`list`*`, `*`flag`*` [, `*`delimiter`*`])`	Sorts the *`list`* in ascending order, using a space or the optional *`delimiter`* as the delimiter between items in the list. The *`flag`* indicates what type of sort to perform: `d` for database reference, `n` for numeric integer, `f` for floating-point, and `a` for alphanumeric.
`space(`*`number`*`)`	Returns a string consisting of the given *`number`* of spaces.
`splice(`*`list1`*`, `*`list2`*`, `*`word`*`)`	Splices the two lists together. If a word in *`list1`* matches the given *`word`*, then the word in corresponding position in *`list2`* is substituted in.
`sqrt(`*`number`*`)`	Returns square root of the *`number`*. The *`number`* may not be negative.
`starttime()`	Returns the `date/time` string when the MUSH was last rebooted.
`stats([`*`player`*`])`	Returns statistics about the MUSH, or the given optional *`player`*, similar to the `@stats` command.

`strlen(string)`	Returns the number of characters in the `string`.
`strmatch(string, pattern)`	Returns `1` if the entire `string` matches the given `pattern`, and `0` otherwise. Not case sensitive.
`sub(a, b)`	Returns `a - b`.
`subj(object)`	Returns the subjective pronoun for the `object` (`he`, `she`, `it`, `they`), based on the `@sex` attribute of the `object`.
`switch(test, action)`	Evaluates the `test`, and then performs the given `action` if the `test` is true; otherwise, it does nothing. The `test` is false if it evaluates to `0`, `-1`, or to the null string. Otherwise, it's true.
`switch(test, action1, action2)`	Evaluates the `test`, and then performs `action1` if `test` is true and `action2` if `test` is false. The `test` is false if it evaluates to `0`, `-1`, or to the null string. Otherwise, it's true.
`tan(number)`	Returns the tangent of the given `number`.
`time()`	Returns the current time on the machine on which the MUSH is running.
`trim(string [, flag [, trim-character]])`	Trims the given `string`, on both sides by default. Leading and trailing spaces are trimmed unless the optional `trim-character` is given. The `flag` can be `b` to trim both sides, `l` to trim the left side, and `r` to trim the right side.
`trunc(floating-point-number)`	Returns a truncated integer version of a `floating-point number`.
`type(object)`	Returns the object type of the `object` (`room`, `exit`, `thing`, or `player`).
`u([object/]attribute)`	Evaluates the given `attribute`.
`ucstr(string)`	Converts the `string` to all uppercase.
`v(argument)`	Returns the value of the variable specified as its `argument`. Some of the possibilities include the following:
`v(0) ... v(9)`	Returns the appropriate stack value.
`v(va) ... v(vz)`	Returns the contents of the appropriate object register.

`v(#)`	Returns the object number of whoever caused the action.
`v(N)`	Return the object name of whoever caused the action, with its first letter capitalized.
`v(n)`	Returns the object name of whoever caused the action.
`v(!)`	Returns the object number of the object calling `v()`.
`v(`*`attribute`*`)`	Returns the value of the appropriate attribute.
`version()`	Returns version information pertaining to the MUSH you are on.
`where(`*`object`*`)`	Returns the "true" location of the *`object`*, which is the actual location if the *`object`* is a player or thing, the source room if the *`object`* is an exit, or `#-1` if the *`object`* is a room.
`wordpos(`*`string`*`, `*`index`*` [,`*`delimiter`*`])`	Returns the number of the word within the given *`string`* where the *`index`* character falls. The words are delimited by either spaces or the optional *`delimiter`*.
`words(`*`string`*`)`	Returns the number of words in the given *`string`*.
`xor(`*`boolean1`*`, `*`boolean2`*`, ..., `*`booleanN`*`)`	Returns `1` if an odd number of the given *`boolean`* values are true, `0` otherwise.

Flag Reference

Table 15.7 shows the various flags that you can set on objects.

Table 15.7. Flag reference.

Code	*Flag `@set` command*	*Meaning*
A	`@set `*`room`*`=ABODE`	Allows room to be a home for players.
a	`@set `*`object`*`=AUDIBLE`	Allows objects to transmit messages inside of them to the outside room. (MUSH only)

Code	*Flag* `@set` *command*	*Meaning*
`B`	`@set` *`player`*`=BUILDER`	Used on MUCKs and sometimes used on MUSHes to indicate the player has building privileges.
`C`	`@set` *`object`*`=CHOWN_OK`	Allows ownership of object to be changed.
`c`		Denotes that a player is currently connected. (MUSH only)
`D`	`@set` *`room`*`=DARK`	Prevents the listing of the room contents.
`D`	`@set` *`object`*`=DARK`	Object does not show up in room contents list.
`d`	`@set` *`object`*`=DESTROY_OK`	Allows object to be destroyed by any player. (MUSH only)
`e`	`@set` *`object`*`=ENTER_OK`	Allows object to be enterable. (MUSH only)
`E`		Denotes an exit.
`F`	`@set` *`room`*`=FLOATING`	Suppresses the disconnected-room message. (MUSH only)
`G`		Used internally for `@destroy`, denotes a room that is set to be destroyed. (MUSH only)
`H`	`@set` *`room`*`=HAVEN`	Disallows player killing in the room.
`h`		Indicates that an object is halted. (MUSH only)
`i`	`@set` *`object`*`=IMMORTAL`	Indicates an object is immortal, so it cannot be killed and does not use up money. Settable only by wizards. (MUSH only)
`J`	`@set` *`room`*`=JUMP_OK`	Allows players to `@teleport` into room.
`K`	`@set` *`object`*`=KEY`	Turns object into a key, which prevents puppets from picking it up. (MUSH only)
`l`	`@set` *`object`*`=LIGHT`	Allows object to be seen in a room that is `DARK`. (MUSH only)
`L`	`@set` *`object`*`=LINK_OK`	Allows links to the object.
`M`	`@set` *`player`*`=MUCKER`	Allows player to create MUF programs in MUCKs. (MUCK only)
`M`	`@set` *`object`*`=MONITOR`	Denotes an object that is a `MONITOR`. (MUSH only)
`m`	`@set me=MYOPIC`	Treats you as if you did not own anything when you use `look` or perform an automatic `look` upon entering a room. (MUSH only)

continues

Table 15.7. continued

Code	*Flag `@set` command*	*Meaning*
N	`@set me=NOSPOOF`	Adds additional output to `@emit` messages, informing you who sent them. (MUSH only)
O	`@set me=OPAQUE`	Prevents other players from seeing what you are carrying; however, the other player can see objects that you are carrying that they own. (MUSH only)
P		Denotes a player.
p	`@set` *`object`*`=puppet`	Turns the object into a puppet. (MUSH only)
Q	`@set me=QUIET`	Prevents you from hearing `set` or `triggered` messages from objects that you own. (MUSH only)
q	`@set me=TERSE`	Prevents you from seeing the description, contents, success or fail messages, and so on, of the rooms that you enter. (MUSH only)
R		Denotes a room.
r		Denotes a robot. (MUSH only)
S	`@set` *`object`*`=STICKY`	Sets object `sticky`. Object goes home when dropped.
S	`@set` *`room`*`=STICKY`	Sets room `sticky`. `Drop-to`'s are delayed.
s	`@set` *`object`*`=SAFE`	Forces you to use the `/override` switch when `@destroy`ing the object. (MUSH only)
T	`@set` *`object`*`=TRACE`	Reports all string substitutions on the object to the object's owner. Useful for debugging. (MUSH only)
t	`@set` *`exit`*`=TRANSPARENT`	Allows a player looking at the exit to see the room description of the room on the other side of the exit, along with the exit's description. (MUSH only)
U	`@set` *`player`*`=UNFINDABLE`	Hides player from `@whereis` command. (MUSH only)
V	`@set` *`object`*`=VISUAL`	Allows any player to see the object's programming. (MUSH only)
v	`@set` *`object`*`=VERBOSE`	Causes all commands executed by the object to be sent to the owner of the object. (MUSH only)

Code	*Flag* `@set` *command*	*Meaning*
`W`	`@set` *`player`*`=wizard`	Denotes a wizard. Settable by wizards only.
`X`	`@set` *`player`*`=SLAVE`	The player may not perform any actions that change the database, nor may any of the player's object's change the database. Settable only by wizards. (MUSH only)
`Y`	`@set` *`object`*`=PARENT_OK`	Any object that passes this object's ParentLock may make this object a parent of any object it controls. (MUSH only)

Summary

In this chapter, you learned how to use and program MUSHes and MUCKs. You learned how to set up your character, build rooms, create objects, and program the MUD.

A

APPENDIX

The MUD Yellow Pages

This appendix lists MUDs and MUD-related resources that can be found on the Internet.

MUDs

This section is a list of over 300 different MUDs. Only LPMUDs, DikuMUDs, MOOs, MUSHes, and MUCKs are shown because they are the primary focus of this book. This MUD list has been compiled from many other lists and sites on the WWW. At the time of this writing, I have personally connected to all the MUDs (and their URLs) listed here. However, the Internet is a rapidly changing entity, and by the time you read this, some of the addresses may have changed. When an address changes, often the MUD will leave a pointer to the new site at the old site . MUDs also tend to be very volatile, as they often are run by their administrators as a hobby. So, do not be too surprised to find that some of the MUDs that appear in this list do not work.

NOTE For a more up-to-date listing, and for direct links to all the MUDs listed here, point your Web browser at `http://www.mcp.com/sams/books/muds/mudlist.html`.

Each entry has the MUD's name, its Internet address in name and number form, its port number, and a MUD type. LPMUD, DikuMUD, MOO, MUSH, and MUCK make up the majority of entries in the MUD type category. The following other MUD types are used:

MudOS—MudOS is a derivative of LPMUD. You likely will not notice many differences, but these MUDs are listed separately for accuracy.

Circle, MERC, ROM—These are different types of MUDs that were derived from the original DikuMUD system.

PERN—Pern is the world in the *Dragonriders of Pern* series by Anne McCaffery, and there are quite a few MUSHes and MOOs based on this world. Rather than writing a description for each MUD designating it as a PERN MUD, /PERN has been added to the end of the MUD type.

ST—The *Storyteller* system is a popular role-playing system used in *The Vampire: The Masquerade, Mage: The Ascension*, and *Werewolf: The Apocalypse*, role-playing games from White Wolf Games Studio. There are many MUSHes set in this environment, which also is called the World of Darkness. Each MUSH is set in a different city and will have a different flavor. If you are familiar with these role-playing games, you probably will enjoy these MUSHes. If you are not familiar with the system, you might want to buy one of the books before you try one of these MUSHes. `/ST` is added to the end of the MUD type to designate these types of MUDs. For more specifics on this type of MUSH try:

`http://blackmagic.com/amy/mush.html.`

Included are URLs for many MUDs. Many of the WWW sites for these MUDs are very informative. They also should help you filter through this list and find MUDs you might enjoy.

Some MUDs listed have a brief description. If the MUD is set in some world that you might recognize, that is mentioned. It also is mentioned if the MUD has a special focus or if it is the original or home MUD for a certain MUD type.

5th Dimension
MUD IP Name Address: `gauss.ifm.liu.se`
IP Number: `130.236.50.9`
Port: `3000`
MUD Type: MudOS

Acer Isle Virtual World
MUD IP Name Address: `cave.pg.md.us`
IP Number: `192.239.102.2`
Port: `2222`
MUD Type: MUSH

ADAMANT
MUD IP Name Address: `rm600.informatik.th-darmstadt.de`
IP Number: `130.83.9.19`
Port: `4711`
MUD Type: LPMUD

After Five
MUD IP Name Address: `toybox.infomagic.com`
IP Number: `165.113.211.2`
Port: `9999`
MUD Type: MUCK
URL: `http://ma.itd.com:8000/afterfive.html`

Age of Legends
MUD IP Name Address: `hub.eden.com`
IP Number: `199.171.21.21`
Port: `6969`
MUD Type: MERC/ROM

Albanian
MUD IP Name Address: `fred.indstate.edu`
IP Number: `139.102.12.14`
Port: `2150`
MUD Type: DikuMUD

AlatiaMUD
MUD IP Name Address: `aann.tyrell.net`
IP Number: `192.175.8.12`
Port: `3000`
MUD Type: LPMUD

AlexMUD
MUD IP Name Address: `alexmud.stacken.kth.se`
IP Number: `130.237.234.3`
Port: `000`
MUD Type: DikuMUD
Comments: This is the original DikuMUD.

Altered DimensionsMUSH 2
MUD IP Name Address: `spruce.evansville.edu`
IP Number: `192.195.225.3`
Port: `6250`
MUD Type: MUSH
URL: `http://www.math.okstate.edu/~russ/muds/amber.html`
Comments: This MUSH is set in the world of Roger Zelzany's *Amber* series. It also uses elements of the *Amber* Diceless Role-Playing Game.

Ancient Anguish
MUD IP Name Address: `ancient.anguish.org`
IP Number: `199.34.48.6`
Port: `2222`
MUD Type: LPMUD
URL: `http://end2.bedrock.com/Ancient_Anguish/aa.html`

Angalon
MUD IP Name Address: `neuromancer.tamu.edu`
IP Number: `128.194.47.9`
Port: `3011`
MUD Type: LPMUD

AngrealMOO
MUD IP Name Address: `j302604012.resnet.cornell.edu`
IP Number: `128.253.150.10`
Port: `9000`
MUD Type: MOO
Comments: This is one of the few MOOs that focus on role-playing. It is a world based on Robert Jordan's *Wheel of Time* series.

Anime MUCK
MUD IP Name Address: `eith.biostr.washington.edu`
IP Number: `128.95.44.29`
Port: `2035`
MUD Type: MUCK

Apex
MUD IP Name Address: `apex.ccs.yorku.ca`
IP Number: `130.63.237.12`
Port: `4201`
MUD Type: MUSH

Apocalypse IV
MUD IP Name Address: `sapphire.geo.wvu.edu`
IP Number: `157.182.168.20`
Port: `4000`
MUD Type: DikuMUD

Arctic
MUD IP Name Address: `arctic.csua.berkeley.edu`
IP Number: `128.32.43.55`
Port: `2700`
MUD Type: DikuMUD

Aurora
MUD IP Name Address: `aurora.etsiig.uniovi.es`
IP Number: `156.35.41.20`

Port: 3000
MUD Type: MudOS

AustinMUD
MUD IP Name Address: imv.aau.dk
IP Number: 130.225.2.2
Port: 4000
MUD Type: DikuMUD

Barren Realms
MUD IP Name Address: liii.com
IP Number: 198.207.193.1
Port: 8000
MUD Type: MERC

BatMUD
MUD IP Name Address: bat.cs.hut.fi
IP Number: 130.223.40.180
Port: 23
MUD Type: LPMUD
URL: http://bat.cs.hut.fi/

BayMOO
MUD IP Name Address: baymoo.sfsu.edu
IP Number: 130.212.41.251
Port: 8888
MUD Type: MOO
URL: http://baymoo.sfsu.edu:4242/

Belior Rising
MUD IP Name Address: brazil-nut.enmu.edu
IP Number: 192.94.216.80
Port: 4301
MUD Type: MUSH/PERN

Bloodletting:Dublin by Night
MUD IP Name Address: piaget.psych.mun.ca
IP Number: 134.153.20.10
Port: 4991
MUD Type: MUSH/ST

Blue Facial MUD
MUD IP Name Address: dallet.channel1.com
IP Number: 199.1.13.9
Port: 1234
MUD Type: MERC

Boo MOO
MUD IP Name Address: `pinot.callamer.com`
IP Number: `199.74.141.2`
Port: `1234`
MUD Type: MOO

Brazilian Dreams
MUD IP Name Address: `red_panda.tbyte.com`
IP Number: `198.211.131.13`
Port: `8888`
MUD Type: MUCK

Camelot MUSH
MUD IP Name Address: `camelot.cit.buffalo.edu`
IP Number: `128.205.3.103`
Port: `5440`
MUD Type: MUSH

Castle D'Image
MUD IP Name Address: `fogey.stanford.edu`
IP Number: `36.22.0.31`
Port: `5555`
MUD Type: MUSH

CaveMUCK
MUD IP Name Address: `cave.tcp.com`
IP Number: `128.95.44.29`
Port: `2283`
MUD Type: MUCK

Chiba Sprawl MOO
MUD IP Name Address: `chiba.picosof.com`
IP Number: `165.227.31.2`
Port: `7777`
MUD Type: MOO
URL: `http://sensemedia.net/sprawl`

Children of the Atom
MUD IP Name Address: `bison.range.orst.edu`
IP Number: `128.193.121.98`
Port: `2099`
MUD Type: MUSH
Comments: This MUSH is based on the *X-men* comic books by Marvel.

Chomestoru
MUD IP Name Address: `dfw.net`
IP Number: `198.175.15.10`
Port: `4000`
MUD Type: DikuMUD

City of Darkness
MUD IP Name Address: `melandra.cs.man.ac.uk`
IP Number: `130.88.240.110`
Port: `2000`
MUD Type: MUSH/ST

Conspiracy
MUD IP Name Address: `almond.enmu.edu`
IP Number: `192.94.216.77`
Port: `1066`
MUD Type: MUSH
URL: `http://www.uunet.ca/~kris/cons.html`

Crossed Swords
MUD IP Name Address: `shsibm.shh.fi`
IP Number: `128.214.44.251`
Port: `3000`
MUD Type: MudOS

CrystalMUSH
MUD IP Name Address: `moink.nmsu.edu`
IP Number: `128.123.8.115`
Port: `6886`
MUD Type: MUSH
Comments: This MUSH is set in the world of Anne McCaffrey's *Crystal Singer* series.

CyberSphere
MUD IP Name Address: `vv.com`
IP Number: `204.183.126.200`
Port: `7777`
MUD Type: MOO
URL: `http://www.vv.com/cs/cybersphere.html`

DaedalusMOO
MUD IP Name Address: `daedalus.com`
IP Number: `198.242.218.1`
Port: `7777`
MUD Type: MOO

Danse Macabre
MUD IP Name Address: `omega.acusd.edu`
IP Number: `192.195.155.207`
Port: `9999`
MUD Type: MUSH/ST

Dark Castle
MUD IP Name Address: foxtrot.rahul.net
IP Number: 192.160.13.6
Port: 6666
MUD Type: DikuMUD

Dark Gift
MUD IP Name Address: sulaco.library.cmu.edu
IP Number: 128.2.21.47
Port: 6250
MUD Type: MUSH/ST

Dark Metal
MUD IP Name Address: pharos.acusd.edu
IP Number: 192.195.155.201
Port: 9999
MUD Type: MUSH

Dark Wind
MUD IP Name Address: darkwind.i-link.com
IP Number: 199.120.94.33
Port: 3000
MUD Type: LPMUD

Darker Realms
MUD IP Name Address: worf.tamu.edu
IP Number: 165.91.112.221
Port: 2000
MUD Type: LPMUD

Darkweb
MUD IP Name Address: steak.nmt.edu
IP Number: 129.138.14.19
Port: 6251
MUD Type: MUSH/ST

Dawn of the Immortals
MUD IP Name Address: immortals.ncsa.uiuc.edu
IP Number: 141.142.26.2
Port: 2000
MUD Type: LPMUD
URL: http://www.shsu.edu/~stdtdc/doti.html

Dawn Sisters
MUD IP Name Address: arms.gps.caltech.edu
IP Number: 131.215.67.3
Port: 9944
MUD Type: MUSH/PERN

Death's Domain
MUD IP Name Address: `cybernet.cse.fau.edu`
IP Number: `131.91.80.79`
Port: `9000`
MUD Type: DikuMUD

Deeper Trouble
MUD IP Name Address: `alk.iesd.auc.dk`
IP Number: `130.225.48.46`
Port: `4242`
MUD Type: LPMUD

Desert Wings
MUD IP Name Address: `jpd.ch.man.ac.uk`
IP Number: `130.88.12.119`
Port: `4201`
MUD Type: MUSH/PERN

Discordia
MUD IP Name Address: `discordia.phya.utoledo.edu`
IP Number: `131.183.60.1`
Port: `4201`
MUD Type: MUSH

Diviniation Web
MUD IP Name Address: `bill.math.uconn.edu`
IP Number: `137.99.17.5`
Port: `9393`
MUD Type: MUCK

Doom MUD
MUD IP Name Address: `neuromancer.hacks.arizona.edu`
IP Number: `128.196.230.12`
Port: `4000`
MUD Type: DikuMUD

Dragon Dawn
MUD IP Name Address: `cashew.enmu.edu`
IP Number: `192.94.216.78`
Port: `2222`
MUD Type: MUSH/PERN

DragonDreams
MUD IP Name Address: `jpd.ch.man.ac.uk`
IP Number: `130.88.12.119`
Port: `4444`
MUD Type: MUSH/PERN

Dragon's Den
MUD IP Name Address: `hellfire.dusers.drexel.edu`
IP Number: `129.25.56.246`
Port: `2222`
MUD Type: LPMUD

DragonFire MUD
MUD IP Name Address: `typo.umsl.edu`
IP Number: `134.124.42.197`
Port: `3000`
MUD Type: LPMUD

DragonMUD
MUD IP Name Address: `satan.ucsd.edu`
IP Number: `132.239.1.7`
Port: `4201`
MUD Type: MUSH
URL: `http://is.rice.edu/~pengle/dragon/welcome.html`

Dragonsfire
MUD IP Name Address: `moo.eskimo.com`
IP Number: `204.122.16.3`
Port: `8888`
MUD Type: MOO/PERN

DreaMOO
MUD IP Name Address: `fiinix.mertronet.com`
IP Number: `198.215.126.2`
Port: `8888`
MUD Type: MOO

DruidMuck
MUD IP Name Address: `moink.nmsu.edu`
IP Number: `128.123.8.115`
Port: `4201`
MUD Type: MUCK

EarthMUD
MUD IP Name Address: `via.nl`
IP Number: `193.78.61.1`
Port: `2222`
MUD Type: MudOS
URL: `http://via.nl:2217/`

Edge of Darkness
MUD IP Name Address: `edge.uccs.edu`
IP Number: `128.198.1.70`

Port: 2001
MUD Type: DikuMUD
URL: http://edge.uccs.edu/

Elements of Paradox
MUD IP Name Address: elof.acc.iit.edu
IP Number: 192.41.245.90
Port: 6996
MUD Type: MudOS

ElysuimMUSH
MUD IP Name Address: zeus.acsd.edu
IP Number: 192.195.155.205
Port: 9999
MUD Type: MUSH/ST

Empire
MUD IP Name Address: einstein.physics.drexel.edu
IP Number: 129.25.1.120
Port: 4000
MUD Type: DikuMUD

Enulal
MUD IP Name Address: istcprj1.u-aizu.ac.jp
IP Number: 163.143.125.114
Port: 2222
MUD Type: LPMUD
URL: http://istcprj1.u-aizu.ac.jp:2224/

EON
MUD IP Name Address: mcmuse.mc.maricopa.edu
IP Number: 140.198.66.28
Port: 8888
MUD Type: MOO

Everdark
MUD IP Name Address: panzer.atomic.com
IP Number: 198.64.213.133
Port: 3000
MUD Type: LPMUD

Fantasia
MUD IP Name Address: betz.biostr.washington.edu
IP Number: 128.35.44.22
Port: 4201
MUD Type: MUSH

Farside
MUD IP Name Address: `mud.atinc.com`
IP Number: `198.138.35.198`
Port: `3000`
MUD Type: MERC
URL: `http://zeus.atinc.com/mud.html`

Fiery
MUD IP Name Address: `fiery.eushc.org`
IP Number: `163.246.96.103`
Port: `4000`
MUD Type: DikuMUD

Final Challenge, The
MUD IP Name Address: `mud.primenet.com`
IP Number: `204.245.0.245`
Port: `4000`
MUD Type: MERC

Final Frontiers—TrekMOO
MUD IP Name Address: `trekmoo.microserve.com`
IP Number: `192.204.120.6`
Port: `2499`
MUD Type: MOO
URL: `http://www.microserve.com/~trek/`
Comments: This MOO is set in the world of *Star Trek*.

Final Realms
MUD IP Name Address: `fr.hiof.no`
IP Number: `158.36.33.52`
Port: `2001`
MUD Type: MudOS

First Light
MUD IP Name Address: `gold.t-informatik.ba-stuttgart.de`
IP Number: `141.31.1.16`
Port: `3000`
URL: `http://www.uni-giessen.de/~gdg3/mud/firstl.html`
MUD Type: LPMUD

FredNet MOO
MUD IP Name Address: `fred.net`
IP Number: `198.76.178.2`
Port: `8888`
MUD Type: MOO

FurryMUCK
MUD IP Name Address: `sncils.cns.edu`
IP Number: `138.74.0.10`
Port: `8888`
MUD Type: MUCK
Comments: This is the most popular MUCK and a popular MUD sex spot. Players are anthropomorphic animals. This must-see MUD often is mentioned in articles about MUDs.

Future Realms—TrekMUSH
MUD IP Name Address: `www.onramp.net`
IP Number: `199.1.11.15`
Port: `1701`
MUD Type: MUSH
Comments: This MUSH is set in the world of *Star Trek*.

Garou
MUD IP Name Address: `cesium.clock.org`
IP Number: `17.255.4.43`
Port: `7000`
MUD Type: MUSH/ST

GateWay
MUD IP Name Address: `idiot.alfred.edu`
IP Number: `149.84.4.1`
Port: `6969`
MUD Type: MudOS

Genesis
MUD IP Name Address: `spica3.cs.chalmers.se`
IP Number: `129.16.227.203`
Port: `3011`
MUD Type: LPMUD
Comments: This is the original LPMUD.

Genocide
MUD IP Name Address: `genocide.shsu.edu`
IP Number: `192.92.115.145`
Port: `2222`
MUD Type: LPMUD/PK
URL: `http://www.shsu.edu/~genlpc/`
Comments: This is one of the first and still one of the most popular MUDs, devoted exclusively to player killing. New wars start on a regular basis. At the beginning of a war, all players are brought to life with equal skills and released into the world to kill each other. There are both team and individual wars.

Glass Dragon, The
MUD IP Name Address: `surf.tstc.edu`
IP Number: `161.109.32.2`
Port: `4000`
MUD Type: DikuMUD
URL: `http://surf.tstc.edu/~dmdurkee/`

GodsHome
MUD IP Name Address: `godshome.solace.mh.se`
IP Number: `193.10.118.131`
Port: `3000`
MUD Type: DGD (LPMUD)

Gohs
MUD IP Name Address: `valhalla.acusd.edu`
IP Number: `192.55.87.27`
Port: `9999`
MUD Type: MUSH

Grimne
MUD IP Name Address: `grimne.pvv.unit.no`
IP Number: `129.241.210.223`
Port: `4000`
MUD Type: DikuMUD
URL: `http://www.pvv.unit.no/~haralde/grimne/`

GypsyMUD
MUD IP Name Address: `hopi.dtcc.edu`
IP Number: `138.123.84.240`
Port: `4000`
MUD Type: DikuMUD

Hall of Fame Mud
MUD IP Name Address: `marvin.df.lth.se`
IP Number: `130.235.88.94`
Port: `2000`
MUD Type: LPMUD

HARI MUD
MUD IP Name Address: `tc0.chem.tue.nl`
IP Number: `131.155.94.3`
Port: `6997`
MUD Type: LPMUD

Harper's Tale
MUD IP Name Address: `srcrisc.srce.hr`
IP Number: `161.53.3.2`
Port: `8888`
MUD Type: MOO/PERN

HAVEN
MUD IP Name Address: `idrz07.ethz.ch`
IP Number: `129.132.76.8`
Port: `1999`
MUD Type: LPMUD

Hero of the Lance 2
MUD IP Name Address: `mud.technet.sg`
IP Number: `192.169.33.110`
Port: `4040`
MUD Type: MERC/ROM

Highlands
MUD IP Name Address: `jedi.cis.temple.edu`
IP Number: `129.32.32.70`
Port: `9001`
MUD Type: MERC

HoloMUCK
MUD IP Name Address: `collatz.mcrcim.mcgill.edu`
IP Number: `132.206.78.1`
Port: `5757`
MUD Type: MUCK

HoloMUD
MUD IP Name Address: `sprawl.fc.net`
IP Number: `198.6.198.6`
Port: `7777`
MUD Type: DikuMUD

Holy Mission
MUD IP Name Address: `alijku05.edvz.uni-linz.ac.at`
IP Number: `140.78.3.1`
Port: `2001`
MUD Type: LPMUD

Hypertext Hotel
MUD IP Name Address: `duke.cs.brown.edu`
IP Number: `128.148.37.8`
Port: `8888`
MUD Type: MOO
URL: `http://duke.cs.brown.edu:8888/`

Idea Exchange, The
MUD IP Name Address: `imaginary.com`
IP Number: `199.199.122.10`
Port: `7890`
MUD Type: MudOS
URL: `http://www.imaginary.com:7885/index.html`

Igor MUD
MUD IP Name Address: `igor.mtek.chalmers.se`
IP Number: `129.16.61.113`
Port: `1701`
MUD Type: DGD (LPMUD)

ImagECastle
MUD IP Name Address: `fogey.stanford.edu`
IP Number: `36.22.0.31`
Port: `4201`
MUD Type: MUSH

Incarnations
MUD IP Name Address: `lumley.cais.com`
IP Number: `204.180.173.1`
Port: `4201`
MUD Type: `MUSH`
Comments: This MUSH is set in the world of Piers Anthony's *Incarnations of Immortality* series.

Ivory Tower
MUD IP Name Address: `marvin.macc.wisc.edu`
IP Number: `192.217.237.7`
Port: `2000`
MUD Type: LPMUD

Jay's House MOO
MUD IP Name Address: `jhm.ccs.neu.edu`
IP Number: `129.10.111.77`
Port: `1709`
MUD Type: MOO
URL: `http://jhm.moo.mud.org:7043/`
Comments: This is a development oriented MOO where many of the latest new MOO concepts are first implemented.

JeenusTooMUD
MUD IP Name Address: `heegaard.mth.pdx.edu`
IP Number: `131.252.40.91`
Port: `4000`
MUD Type: DikuMUD

Jurassic Weyr
MUD IP Name Address: `adamwest.ins.cwru.edu`
IP Number: `129.22.8.52`
Port: `6250`
MUD Type: MUSH/PERN

KallistiMUD
MUD IP Name Address: `jadzia.peak.org`
IP Number: `198.68.23.23`
Port: `4000`
MUD Type: DikuMUD

KAOS MUD
MUD IP Name Address: `flower.aud.temple.edu`
IP Number: `155.247.42.7`
Port: `4000`
MUD Type: DikuMUD

Kerovnia
MUD IP Name Address: `atlantis.edu`
IP Number: `204.97.113.150`
Port: `1984`
MUD Type: LPMUD

KoBra Mud
MUD IP Name Address: `kobra.et.tudelft.nl`
IP Number: `130.161.144.236`
Port: `23`
MUD Type: LPMUD

LambdaMOO
MUD IP Name Address: `lambda.xerox.com`
IP Number: `192.216.54.2`
Port: `8888`
MUD Type: MOO
Comments: The MOO of MOOs, a must-see just because it is LambdaMOO. This is mentioned in all articles that appear in the popular press that talk about MUDs.

Lands of Tabor, The
MUD IP Name Address: `n/a`
IP Number: `165.95.7.122`
Port: `9999`
MUD Type: LPMUD

Last Outpost
MUD IP Name Address: `lo.millcomm.com`
IP Number: `199.170.133.6`
Port: `4000`
MUD Type: DikuMUD

Legend of the Winds
MUD IP Name Address: `ccsun44.csie.nctu.edu.tw`
IP Number: `140.113.17.168`
Port: `4040`
MUD Type: MERC

Loch Ness
MUD IP Name Address: `armageddon.imp.ch`
IP Number: `157.161.1.15`
Port: `2222`
MUD Type: MudOS

Looney
MUD IP Name Address: `cp.tn.tudeflt.nl`
IP Number: `192.31.126.102`
Port: `8888`
MUD Type: LPMUD

Lost Mud, The
MUD IP Name Address: `goofy.cc.utexas.edu`
IP Number: `128.83.108.24`
Port: `6666`
MUD Type: LPMUD

Lost Souls
MUD IP Name Address: `lostsouls.desertmoon.com`
IP Number: `198.102.68.58`
Port: `3000`
MUD Type: LPMUD
URL: `http://lostsouls.desertmoon.com/`

LustyMud
MUD IP Name Address: `lusty.tamu.edu`
IP Number: `128.194.9.199`
Port: `2000`
MUD Type: LPMUD

MadROM
MUD IP Name Address: `hector.turing.toronto.edu`
IP Number: `128.100.5.10`
Port: `1536`
MUD Type: MERC/ROM
URL: `http://www.io.org/~sofa`

Marches of Antan
MUD IP Name Address: `checfs2.ucsd.edu`
IP Number: `132.239.68.9`
Port: `3000`
MUD Type: MudOS

Masquerade, The
MUD IP Name Address: phobos.unm.edu
IP Number: 129.24.8.3
Port: 4444
MUD Type: MUSH/ST

Medieva Cyberspace
MUD IP Name Address: medievia.netaxs.com
IP Number: 198.69.186.36
Port: 4000
MUD Type: DikuMUD

Metaverse
MUD IP Name Address: metaverse.io.com
IP Number: 199.170.88.12
Port: 7777
MUD Type: MOO
Comments: This MOO is run by Steve Jackson Games and is a good resource for people into role-playing games.

Midnight Sun
MUD IP Name Address: midnight-sun.ludd.luth.se
IP Number: 130.240.16.23
Port: 3000
MUD Type: LPMUD
URL: http://mud.ludd.luth.se:80/midnight/

Might, Magic & Mushrooms
MUD IP Name Address: prime.mdata.fi
IP Number: 192.98.43.2
Port: 6047
MUD Type: DGD

MirrorMOO
MUD IP Name Address: mirror.ccs.neu.edu
IP Number: 129.10.112.76
Port: 8889
MUD Type: DGD (MOO)

MoonStar
MUD IP Name Address: pulsar.hsc.edu
IP Number: 192.135.84.5
Port: 4321
MUD Type: LPMUD

MooseHead SLED II
MUD IP Name Address: eskinews.eskimo.com
IP Number: 204.122.16.44
Port: 4000
MUD Type: DikuMUD

Mortal Realms
MUD IP Name Address: `hydrogen.ee.utulsa.edu`
IP Number: `129.244.42.48`
Port: `4321`
MUD Type: MERC

Muddy Waters
MUD IP Name Address: `hot.caltech.edu`
IP Number: `131.215.9.49`
Port: `3000`
MUD Type: LPMUD

MuMOO
MUD IP Name Address: `chestnut.enmu.edu`
IP Number: `192.94.216.74`
Port: `7777`
MUD Type: MOO

Mystic Adventure
MUD IP Name Address: `miniac.etu.gel.ulaval.ca`
IP Number: `132.203.14.100`
Port: `4000`
MUD Type: DikuMUD

NAILS
MUD IP Name Address: `flounder.rutgers.edu`
IP Number: `128.6.128.5`
Port: `5150`
MUD Type: MUCK

NamelessMUSH
MUD IP Name Address: `occams.dfci.harvard.edu`
IP Number: `134.174.51.13`
Port: `6666`
MUD Type: MUSH

NannyMUD
MUD IP Name Address: `birka.lysator.liu.se`
IP Number: `130.236.254.159`
Port: `2000`
MUD Type: LPMUD

NANVAENT
MUD IP Name Address: `corrour.cc.strath.ac.uk`
IP Number: `130.159.220.8`
Port: `3000`
MUD Type: MudOS
URL: `http://aragorn.uio.no/nanvaent/`

NecroMOO
MUD IP Name Address: `cyberion.bbn.com`
IP Number: `128.89.2.139`
Port: `4242`
MUD Type: MOO

NES Mush
MUD IP Name Address: `snowhite.ee.pdx.edu`
IP Number: `131.252.10.66`
Port: `9999`
MUD Type: MUSH
Comments: This MUSH is based on *The Never-Ending Story*.

New Hercules MUD
MUD IP Name Address: `sunshine.eushc.edu`
IP Number: `163.246.96.102`
Port: `3000`
MUD Type: DikuMUD

New Moon
MUD IP Name Address: `jove.cs.pdx.edu`
IP Number: `131.252.21.12`
Port: `7680`
MUD Type: MudOS

Nightmare
MUD IP Name Address: `nightmare.imaginary.com`
IP Number: `199.199.122.10`
Port: `1701`
MUD Type: MudOS
URL: `http://www.imaginary.com:1696/Nightmare/Nightmare.html`

Nirvana
MUD IP Name Address: `elof.acc.iit.edu`
IP Number: `192.41.245.90`
Port: `3500`
MUD Type: LPMUD
URL: `http://craig.stanford.edu/Nirvana/home.html`

Northern Crossroads
MUD IP Name Address: `ugsparc21.eecg.toronto.edu`
IP Number: `128.100.13.101`
Port: `9000`
MUD Type: DikuMUD

Nuclear War
MUD IP Name Address: `melba.astrakan.hgs.se`
IP Number: `130.238.206.12`

Port: 23
MUD Type: LPMUD
URL: http://www.astrakan.hgs.se/nuke/nuke.html

OpalMUD
MUD IP Name Address: opal.cs.virginia.edu
IP Number: 128.143.60.14
Port: 4000
MUD Type: DikuMUD

Other MUSH
MUD IP Name Address: pebkac.satelnet.org
IP Number: 198.30.149.9
Port: 4201
MUD Type: MUSH

Overdrive
MUD IP Name Address: castor.acs.oakland.edu
IP Number: 141.210.10.109
Port: 5195
MUD Type: MudOS

PaderMUD
MUD IP Name Address: mud.uni-paderborn.de
IP Number: 131.234.12.13
Port: 3000
MUD Type: DGD (LPMUD)

Paradox
MUD IP Name Address: adl.uncc.edu
IP Number: 152.15.15.18
Port: 10478
MUD Type: LPMUD

Pattern, The
MUD IP Name Address: epsilon.me.chalmers.se
IP Number: 129.16.50.30
Port: 6047
MUD Type: DGD
Comments: This is the home MUD for Dworkin's Generic Driver (DGD). It is the best place to go to ask DGD-related questions or to learn more about DGD.

Patternfall
MUD IP Name Address: misc.acf.nyu.edu
IP Number: 128.122.207.19
Port: 4444
MUD Type: MUSH
Comments: Another MUSH based on Roger Zelazny's *Amber* series.

Perilous Realms
MUD IP Name Address: pr.mese.com
IP Number: 155.229.1.4
Port: 23
MUD Type: DikuMUD
Comments: This is a very customized version of DikuMUD. It's hard to find many similarities to the standard DikuMUD set-up.

PernMUSH
MUD IP Name Address: astral.magic.ca
IP Number: 199.166.230.69
Port: 4201
MUD Type: MUSH/PERN

pHANTAZM
MUD IP Name Address: fpa.com
IP Number: 198.242.217.1
Port: 4000
MUD Type: Circle

Phidar
MUD IP Name Address: cdsgw.crystaldata.com
IP Number: 198.49.103.129
Port: 9000
MUD Type: MERC/ROM

Phoenix
MUD IP Name Address: albert.bu.edu
IP Number: 128.197.74.10
Port: 3500
MUD Type: MudOS

PiliusMUD
MUD IP Name Address: hopi.dtcc.edu
IP Number: 38.123.84.240
Port: 5757
MUD Type: Circle

PK MUD
MUD IP Name Address: kennedy.ecn.uoknor.edu
IP Number: 129.15.112.38
Port: 5000
MUD Type: DikuMUD/PK

PMC-MOO
MUD IP Name Address: hero.village.virginia.edu
IP Number: 128.143.200.59
Port: 7777
MUD Type: MOO

PrairieMUSH
MUD IP Name Address: `prairienet.org`
IP Number: `192.17.3.3`
Port: `4201`
MUD Type: MUSH

PrimalMUD
MUD IP Name Address: `jeack.apana.org.au`
IP Number: `202.12.87.82`
Port: `4000`
MUD Type: Circle

Psycho-thriller
MUD IP Name Address: `atlantis.edu`
IP Number: `204.97.113.150`
Port: `3000`
MUD Type: `LPMUD/PK`

Quovadis
MUD IP Name Address: `mud.imp.ch`
IP Number: `157.161.1.10`
Port: `2345`
MUD Type: LPMUD

Ragnarok
MUD IP Name Address: `rag.com`
IP Number: `192.108.254.22`
Port: `2222`
MUD Type: LPMUD
URL: `http://www.rag.com/ or http://ragnarok.teleport.com/`

Realm of Magic
MUD IP Name Address: `p106.informatik.uni-bremen.de`
IP Number: `134.102.216.7`
Port: `4000`
MUD Type: Circle
Comments: This DikuMUD was used as a reference throughout this book (for DikuMUD-related material).

Realms of Imagination
MUD IP Name Address: `foxtrot.rahul.net`
IP Number: `192.160.13.6`
Port: `4000`
MUD Type: DikuMUD

Realms of the Dragon
MUD IP Name Address: `cm-u04.umd.umich.edu`
IP Number: `141.215.69.7`
Port: `3000`
MUD Type: MudOS

RealmsMUCK
MUD IP Name Address: `eith.biostr.washington.edu`
IP Number: `128.95.44.29`
Port: `7765`
MUD Type: MUCK

RealmsMUD
MUD IP Name Address: `realms.dorsai.org`
IP Number: `198.3.127.200`
Port: `1501`
MUD Type: LPMUD
Comments: This LPMUD was used as a reference throughout this book.

Regenesis
MUD IP Name Address: `birka.lysator.liu.se`
IP Number: `130.236.254.159`
Port: `7475`
MUD Type: LPMUD

Renegade Outpost
MUD IP Name Address: `mercury.cnct.com`
IP Number: `165.254.118.47`
Port: `9999`
MUD Type: DikuMUD

Requiem
MUD IP Name Address: `gangrel.hybrid.com`
IP Number: `198.13.9.3`
Port: `2030`
MUD Type: MUSH/ST
URL: `http://www.best.com/~wyldefyr/requiem.html`

Revenge of End of the Line
MUD IP Name Address: `mud.stanford.edu`
IP Number: `36.21.0.99`
Port: `2010`
MUD Type: LPMUD

Rift
MUD IP Name Address: `cave.pg.md.us`
IP Number: `19.239.102.2`
Port: `4444`
MUD Type: MUSH
Comments: This MUSH is set in the worlds of Raymond E. Feist's *Riftwars* saga.

RockyMud
MUD IP Name Address: `hermes.dna.mci.com`
IP Number: `166.41.48.146`
Port: `4000`
MUD Type: DikuMUD

Rogue
MUD IP Name Address: `rogue.coe.ohio-state.edu`
IP Number: `128.146.144.12`
Port: `2222`
MUD Type: LPMUD/PK

RoninMUD
MUD IP Name Address: `ronin.bchs.uh.edu`
IP Number: `129.7.2.127`
Port: `5000`
MUD Type: DikuMUD
URL: `http://lsma28.jsc.nasa.gov/`

Sanctuary
MUD IP Name Address: `pauli.sos.clarkson.edu`
IP Number: `128.153.32.10`
Port: `9000`
MUD Type: DikuMUD

Sanguinis Nobilis
MUD IP Name Address: `colossus.acusd.edu`
IP Number: `192.195.155.200`
Port: `4444`
MUD Type: MUSH/ST

Shadowdale
MUD IP Name Address: `dale.hsc.unt.edu`
IP Number: `129.120.104.40`
Port: `7777`
MUD Type: DikuMUD

Shadowrun MUSH
MUD IP Name Address: `picard.dnaco.net`
IP Number: `204.95.80.4`
Port: `4201`

MUD Type: MUSH
URL: `http://jhm.moo.mud.org:7043/`
URL: `http://www.dnaco.net/~shadow/`
Comments: This MUSH is set in the world of FASA's *Shadowrun* role-playing game series.

Shadow's Edge
MUD IP Name Address: `wubba.lowe.org`
IP Number: `192.195.202.22`
Port: `4000`
MUD Type: LPMUD

Shards
MUD IP Name Address: `vesta.unm.edu`
IP Number: `129.24.120.253`
Port: `7777`
MUD Type: MUSH/PERN

Silicon Realms
MUD IP Name Address: `sampan.ee.fit.edu`
IP Number: `163.118.30.9`
Port: `4000`
MUD Type: DikuMUD

Sloth II
MUD IP Name Address: `ai.cs.ukans.edu`
IP Number: `129.237.80.113`
Port: `6101`
MUD Type: DikuMUD

SouCon
MUD IP Name Address: `beechnut.enmu.edu`
IP Number: `192.94.216.86`
Port: `4201`
MUD Type: MUSH/PERN

SPAM MUD
MUD IP Name Address: `ganymede.ics.uci.edu`
IP Number: `128.195.10.9`
Port: `5000`
MUD Type: MERC/ROM

Split Second
MUD IP Name Address: `lestat.shv.hb.se`
IP Number: `193.10.174.40`
Port: `3000`
MUD Type: LPMUD

StackMUD
MUD IP Name Address: marcel.stacken.kth.se
IP Number: 130.237.234.17
Port: 8000
MUD Type: MERC

StickMUD
MUD IP Name Address: lancelot.cc.jyu.fi
IP Number: 130.234.40.4
Port: 7680
MUD Type: LPMUD

Stick in the MUD
MUD IP Name Address: ugsparc31.eecg.utoronto.ca
IP Number: 128.100.13.111
Port: 9000
MUD Type: MERC/ROM

StrikeNet
MUD IP Name Address: mozart.fin.depaul.edu
IP Number: 140.192.40.5
Port: 4000
MUD Type: DikuMUD

STYX
MUD IP Name Address: dreamtime.nmsu.edu
IP Number: 128.123.8.116
Port: 3000
MUD Type: MudOS

Sword Quest
MUD IP Name Address: kennedy.ecn.uoknor.edu
IP Number: 129.15.112.38
Port: 5500
MUD Type: DikuMUD

SwordsMUSH
MUD IP Name Address: world.std.com
IP Number: 192.74.137.5
Port: 4201
MUD Type: MUSH
Comments: This MUSH is set in the world of Fred Saberhagen's *Book of Swords* series.

Tapestries
MUD IP Name Address: tapestries.tcp.com
IP Number: 128.95.44.29
Port: 2069
MUD Type: MUCK

TAPPMUD
MUD IP Name Address: surprise.pro.ufz.de
IP Number: 141.65.40.11
Port: 6510
MUD Type: LPMUD

Temple of Syrinx MOO
MUD IP Name Address: cimsun.aidt.edu
IP Number: 192.211.38.63
Port: 2112
MUD Type: MOO

Terminal Guidance MUD
MUD IP Name Address: shire.ncsa.uiuc.edu
IP Number: 141.142.103.6
Port: 6969
MUD Type: DikuMUD

Tesseract
MUD IP Name Address: mud.ior.com
IP Number: 199.79.239.13
Port: 9000
MUD Type: DikuMUD/ROM

Texas Twilight
MUD IP Name Address: seds.lpl.arizona.edu
IP Number: 128.196.64.66
Port: 6250
MUD Type: MUSH/ST

Thunderdome II
MUD IP Name Address: tdome.montana.edu
IP Number: 199.2.139.3
Port: 5555
MUD Type: Circle

Timewarp
MUD IP Name Address: quark.gmi.edu
IP Number: 192.138.137.39
Port: 5150
MUD Type: LPMUD

TinyCWRU
MUD IP Name Address: caisr2.caisr.cwru.edu
IP Number: 129.22.24.22
Port: 4201
MUD Type: MUSH

TinyTIM
MUD IP Name Address: `yay.tim.org`
IP Number: `155.37.1.251`
Port: `5440`
MUD Type: MUSH
URL: `http://yay.tim.org/TIMhome.html`

TMI-2
MUD IP Name Address: `tmi-2.ccs.neu.edu`
IP Number: `129.10.114.86`
Port: `5555`
MUD Type: MUD
URL: `http://tmi.lp.mud.org:5550/`
Comments: This is the home of MudOS. TMI is short for The MUD Institute, and it is a good place to go to ask questions relating to LPC and programming LPMUDs and MudOS MUDs.

ToonMUSH 3
MUD IP Name Address: `brahe.phys.unm.edu`
IP Number: `198.59.169.11`
Port: `9999`
MUD Type: MUSH

ToonMUSH 4
MUD IP Name Address: `occams.dfci.harvard.edu`
IP Number: `134.174.51.13`
Port: `7777`
MUD Type: MUSH

TrekMUSE
MUD IP Name Address: `grimmy.cnidr.org`
IP Number: `128.109.179.14`
Port: `1701`
MUD Type: `MUSH`
URL: `http://grimmy.cnidr.org/`
Comments: This MUSH is set in the world of *Star Trek*.

Trinity MUSH
MUD IP Name Address: `nomadd.fiu.edu`
IP Number: `131.94.66.12`
Port: `4201`
MUD Type: MUSH/ST

TrippyMUSH
MUD IP Name Address: `pebkac.satelnet.org`
IP Number: `198.30.149.9`
Port: `7567`
MUD Type: MUSH

TRON
MUD IP Name Address: `polaris.king.ac.uk`
IP Number: `141.241.84.65`
Port: `3000`
MUD Type: LPMUD

Tsunami
MUD IP Name Address: `adept.csse.muroran-it.ac.jp`
IP Number: `157.19.133.2`
Port: `7680`
MUD Type: MudOS

TUBMUD
MUD IP Name Address: `morgen.cs.tu-berlin.de`
IP Number: `130.149.19.20`
Port: `7680`
MUD Type: LPMUD

Turf
MUD IP Name Address: `teaching6.physics.ox.ac.uk`
IP Number: `163.1.245.206`
Port: `4000`
MUD Type: MERC
URL: `http://sable.ox.ac.uk/~microsoc/turf.html`

Twilight MUD
MUD IP Name Address: `boreal.alfred.edu`
IP Number: `149.84.33.4`
Port: `1234`
MUD Type: MERC

Two Moons
MUD IP Name Address: `lupine.org`
IP Number: `204.97.2.40`
Port: `4201`
MUD Type: MUSH
URL: `http://www.lupine.org/TwoMoons/Title.html`
Comments: This MUSH is set in the world of *ElfQuest*.

Unbridled Desire
MUD IP Name Address: `epona.magibox.net`
IP Number: `199.171.80.3`
Port: `8888`
MUD Type: MUCK

University of MOO
MUD IP Name Address: `moo.cs.uwindsor.ca`
IP Number: `137.207.192.76`

Port: 7777
MUD Type: MOO

Unsafe Haven
MUD IP Name Address: sibylline.com
IP Number: 199.2.240.10
Port: 4000
MUD Type: Circle
URL: http://www.primenet.com/~stormie/mud.html

Valhalla
MUD IP Name Address: valhalla.com
IP Number: 192.187.153.1
Port: 2444
MUD Type: LPMUD

Valhalla MOO
MUD IP Name Address: valhalla.acusd.edu
IP Number: 192.55.87.27
Port: 4444
MUD Type: MOO

Valhalla MUD
MUD IP Name Address: mud.stacken.kth.se
IP Number: 130.237.234.3
Port: 4242
MUD Type: DikuMUD

Veil of Seduction
MUD IP Name Address: purple.cybernetics.net
IP Number: 198.80.48.9
Port: 6250
MUD Type: MUSH/ST

Vertigo
MUD IP Name Address: vertigo.apana.org.au
IP Number: 203.0.12.1
Port: 2001
MUD Type: MudOS

VIE-MUD
MUD IP Name Address: mud.cs.odu.edu
IP Number: 128.82.6.30
Port: 4000
MUD Type: DikuMUD

Viking
MUD IP Name Address: `viking.pvv.unit.no`
IP Number: `129.241.190.14`
Port: `2001`
MUD Type: MudOS
URL: `http://www.pvv.unit.no/viking/`

Virtual World of Magma
MUD IP Name Address: `magma.leg.ufrj.br`
IP Number: `146.164.70.193`
Port: `4000`
MUD Type: Circle

Void, The
MUD IP Name Address: `rosebud.umiacs.umd.edu`
IP Number: `128.8.120.103`
Port: `4000`
MUD Type: Circle

WackoMUSH
MUD IP Name Address: `red-branch.mit.edu`
IP Number: `18.199.0.29`
Port: `6003`
MUD Type: MUSH

Wayne's World
MUD IP Name Address: `drake.eushc.org`
IP Number: `163.246.32.100`
Port: `9000`
MUD Type: DikuMUD

Windy MUD
MUD IP Name Address: `bitsy.apana.org.au`
IP Number: `203.2.134.3`
Port: `2000`
MUD Type: LPMUD

WriteMUSH
MUD IP Name Address: `palmer.sacc.colostate.edu`
IP Number: `129.82.169.4`
Port: `6250`
MUD Type: MUSH
Comments: This MUSH is for the teaching and discussion of writing.

Zen MOO
MUD IP Name Address: `chesire.cc.oxy.edu`
IP Number: `134.69.1.253`
Port: `777`
MUD Type: MOO

Zombie
MUD IP Name Address: `linux1.sjoki.uta.fi`
IP Number: `192.98.82.5`
Port: `3333`
MUD Type: LPMUD

Mail-Order MUD List

You can get an e-mail subscription to the Doran MUD list by mailing **`mudlist@lore.calpoly.edu`** with **`SUBSCRIBE`** as the subject. If you just want to see the latest copy and not subscribe, use **`ISSUE`** in the subject line. These should be capitalized. The Doran MUD list appears to be one of the most up-to-date and complete.

Usenet Newsgroups Relating to MUDs

This section talks about the Usenet newsgroups that relate to MUDs and includes a brief summary of each.

rec.games.mud.admin

This newsgroup is devoted to administrative issues of MUDs. For example, you often will find postings here when MUDs are looking for new wizards; however, you also might check in the newsgroups that relate specifically to the MUDs in which you are interested. You also will see many postings asking for machines where people can run their MUDs.

rec.games.mud.announce

This newsgroup is moderated and contains informational articles on MUDs, as well as announcements about new MUD openings, MUD closings, and other general informational announcements.

rec.games.mud.diku

This newsgroup is devoted to DikuMUD and its derivatives (CircleMUD, MERC, and so on).

rec.games.mud.lp

This newsgroup is devoted to LPMUD and its derivatives (MudOS) and related topics (LPMUD MUDlibs).

rec.games.mud.misc

This newsgroup is for MUD-related topics that do not fall into other categories. For example, MUDs that do not fall into another category probably will be discussed here.

rec.games.mud.tiny

This newsgroup is for TinyMUD and its derivatives (MUSHes, MUCKS, and MOOs).

NOTE

When posting on one of these newsgroups, please make sure it is the appropriate group. For example, do not post a question about LPC (the programming language of LPMUDs) on `rec.games.mud.tiny`—post it on `rec.games.mud.lp`.

Other Net News

There is a second tier of Net news in the `alt.*` hierarchy. It is very easy to create `alt` newsgroups (whereas the `rec` newsgroups are organized and controlled), and hence, there often are new groups created every day and groups also go away on a regular basis. `alt` newsgroups tend to have a lot more noise and a lot less focus than `rec` newsgroups. If you are interested, search periodically for MUD or MOO and perhaps you will find new groups as they are created. These groups existed at the time of this writing, but may not exist when you read this.

alt.mud

This newsgroup is fairly active and has a lot of MUD-related announcements as well as "where are you" or "where is this MUD" type of inquiries.

alt.flame.mud

This newsgroup appears to be related to flaming MUDs. (Flaming is basically bashing or negative talk about something or somebody.)

MUD Resources on the World Wide Web

The World Wide Web is the fastest growing part of the Internet. It has a very good system for organizing information. As such, there is a lot of MUD-related information on WWW. This section introduces MUD-related sites and provides a brief description of each site.

General MUD Resources on the World Wide Web

This section lists sites that provide very general MUD information and are good WWW starting points for MUD resources.

Hyperlinked Version of the MUD Frequently Asked Questions List

`http://math.okstate.edu/~jds/mudfaqs.html`

The MUD FAQ that has been quoted several times in this book is available here in a hyperlinked format.

The MOO-Cows Frequently Asked Question List

`http://www.ccs.neu.edu/home/fox/moo/moofaq.html`

In a question and answer format, this FAQ has some hyperlinks and also answers to many MOO technical questions and programming issues. This FAQ also has a general section for more basic MOO questions and probably is a good place to start if you want to set up a MOO.

The MOO Help System in WWW Format

`http://jh.ccs.neu.edu:7043/help/subject!summary`

This MOO is a good way to familiarize yourself with MOO commands. While not completely hyperlinked, this server can be good for navigating through the MOO help system in a hierarchical format so that you can take a look at all the commands in each category.

The MUD Resource Collection from Lydia Leong

`http://www.cis.upenn.edu/~lwl/mudinfo.html`

This collection is a comprehensive resource for MUD-related information. This site is a great starting point for those interested in learning more about the MUD via the WWW. This site has a large list of pointers to other MUD-related WWW sites. Although it covers all MUD types, it has a wide selection of MUSH-related material.

Fran Litterio's Multi-User Dungeon Page

`http://draco.centerline.com:8080/~franl/mud.html`

This is another complete, general MUD site. This site has links to many MUD-related pages, but its focus leans a little more towards MOOs.

The *aragon.uio.no* WWW Server for MUD Information

`http://aragorn.uio.no/`

Dedicating itself to being the WWW MUD starting point, this site has many links to other MUD information, with a focus on LPMUD-related links.

MUDS @ Lysator

`http://www.lysator.liu.se:7500/mud/main.html`

This is a generic MUD page with pointers to a lot of other MUD info. It includes a link (`http://www.lysator.liu.se:7500/mud/The_Dragon_ate...html`) to a *Wired* article on MUDs.

The MUDdex

`http://www.ccs.neu.edu/home/lpb/muddex.html`

This is a good cross-section of MUD information. This site provides an eclectic, but interesting, assortment of MUD information. A special focus is given to MUD history.

Introduction to MU*s

`http://www.vuw.ac.nz/who/Jamie.Norrish/mud/mud.html`

This is a nice collection of interesting Web pages that have to do with MUDs. Many of the links are theoretical in nature, regarding the development of combat systems and the level of realism. There are many documents on MUD design.

Lists of MUDs in a WWW Format

Following are the most popular MUD lists that you can browse or search via the World Wide Web.

Cardiff's MUD Page

`http://www.cm.cf.ac.uk:80/User/Andrew.Wilson/MUDlist/`

The actual page for searching for MUDs is

`http://www.cm.cf.ac.uk:80/htbin/AndrewW/MUDlist/mud_list_search?DORANS`

This site has links to other MUD-related sites, but its main attraction is the searchable MUD list. You can search for MUDs by name or type. Based on the Doran MUD list, this site has a large database MUD and is a good place to go if you need to find the address of a MUD, or if you just want to check out what new MUDs are out there.

Scott Geiger's MUD List

`http://b63062.STUDENT.cwru.edu:80/~mudlist/mud/list.html`

This is a large MUD list compiled and maintained by Scott Geiger. This list has over 500 entries and is updated regularly.

The Nightmare LPMUD List

`http://nightmare.imaginary.com:1696/gateways/mudlist`

This list is a dynamically maintained list of LPMUDs that are part of the LPMUD/MudOS InterMUD communications systems. It only displays MUDs that are currently up.

The Almost-Complete List of MUSHes

`http://www.cis.upenn.edu/~lwl/muds.html`

Another Lydia Leong creation, this is a large list of MUSHes with a brief description of each and a telnet hyperlink for each MUSH. Also, each entry is color-coded to designate the theme.

RiffRaff's Unofficial List of DikuMUD Home Pages

`http://www.webcom.com/~feline/mudhome.html`

This is a list of DikuMUDs with WWW pages.

MUD Technical and Programming Information

This section provides pointers on technical and programming information about MUDs that can be accessed via the World Wide Web.

LPC Guide

`http://www.lysator.liu.se:7500/mud/lpc.html`

This is a hyperlinked guide to LPC, the programming language used on LPMUDs. It includes hyperlinks to the list of LPC `efuns`, other LPC programming information, and the Beginner's and Intermediate LPC Manuals by Descartes of Borg.

The MudOS Manual Pages in WWW Format

`http://aragorn.uio.no/nanvaent/manpages/`

See the MudOS manual pages (online help) in a hyperlinked format. MudOS (an LPMUD derivative) uses a variant of LPC.

The LambdaMOO Programmer's Manual—Table of Contents

`ftp://parcftp.xerox.com/pub/MOO/ProgrammersManual.texinfo_toc.html`

The first site is often busy, so here is an alternate site:

`http://keck.tamu.edu/cgi/MOO/ProgrammersManual.texinfo_toc.html`

This is Pavel Curtis' LambdaMOO Programmer's Manual in a hyperlinked format. This is the most well-known programming reference for LambdaMOO development.

The CircleMUD Home Page

`http://www.cs.jhu.edu/other/jelson/circle.html`

This WWW page is a resource for CircleMUD information. CircleMUD is a derivative MUD driver that was developed from the original DikuMUD code.

Academic Papers Concerning MUDs

MUDs are powerful and can be used for many things other than games. This section lists WWW links for pages discussing other uses of MUDs and academic papers on MUDs.

SunSITE: Papers: Communications

`http://sunsite.unc.edu/dbarberi/comm-papers.html`

Part of the massive SunSITE WWW server, this particular page has links to and abstracts for many papers having to do with MUDs, MOOs, IRC, and other virtual communities. If you are interested in more scholarly thoughts on these types of environments, the papers here would be a good start.

The Lost Library of MOO

`http://lucien.berkeley.edu/moo.html`

This site has pointers to many MOO- and MUD-related research papers. It also has pointers to other archives of papers. Unfortunately, many of the papers pointed to by this site are only available in Postscript format.

Collaborative Networked Communication: MUDs as Systems Tools

`http://www.ccs.neu.edu/home/django/docs/cncmast.html`

This is a paper that discusses the use of MUDs as a tool for corporate and academic information systems. This is a good paper to read if you want to see some of the potential uses of MUDs outside of gaming and basic socializing.

The Integration of MUDs and the World Wide Web

As the World Wide Web has grown in popularity, the idea of integrating MUDs and the World Wide Web has received some attention. The sites that follow point to information and research on this topic.

MOO-WWW Research Directory

`http://www.maths.tcd.ie/pub/mud/moo-www/rdir/rd.html`

This is a list that covers many of the WWW pages available relating to the integration of WWW and MOOs.

WWW MUD Implementations

`http://www.ccs.neu.edu/home/nop/mudwww.html`

This site has pointers to and a brief discussion of two MOOs (Jay's House MOO and Cardiff MOO) that have extensions that allow growing of the MOOs through a WWW interface. This is a good page for learning more about the merging of WWW and MUDs.

Phoenix: A Web/MOO Client

`http://www.bsd.uchicago.edu/Staff/Web_Notes/MOO-overview.html`

This is a discussion of another interface to MOOs through the WWW. This site talks a lot about BioMOO, which is a MOO specifically for biologists and with an experimental WWW interface.

Technical Documentation on web2mush

`http://nimbus.som.cwru.edu/~glenn/mush/tech.html`

This is very detailed documentation, with graphical diagrams, of the web2mush interface. It includes diagrams of how TinyMUSH and the WWW work, and explains the interface web2mush provides between them. It also points to TinyCWRU (`http://nimbus.som.cwru.edu/~glenn/mush/TinyCWRU.html`), which is a WWW-based MUSH. It uses hyperlinks for navigation.

mmMOO

`http://www.peg.apc.org/~firehorse/mmm/mmm.html`

This contains a lot of interesting information on MOOs. This site talks about LambdaMOO and has a *What is a MOO?* section. The site, however, concentrates primarily on the possibility of developing multimedia MOOs.

B

APPENDIX

MUD Glossary

This appendix familiarize you with terms that you probably will encounter while MUDding.

Acronyms

Typing takes a lot of work and MUDders have developed a sort of shorthand using acronyms for some words and phrases that are used on a regular basis.

AC—Armor Class

AFK—Away from the Keyboard. The person playing is leaving the computer for a while, perhaps to have a drink or a smoke.

BF—Boyfriend

BFD—An offensive way of saying Big Frigging Deal

BRB—Be Right Back (see *AFK*)

BRT—Be Right There. An indicator that the character is in transit (usually used in `tells` and `shouts` to let people know you are on your way to meet them)

BTW—By The Way

EP—Experience Points

F2F—Face to face

GF—Girlfriend

HP—Hit Points

IC—In Character; completely immersed in the virtual world and playing the MUD persona

IHHO—In His (or Her) Humble Opinion

IMHO—In My Humble Opinion

IRL—In Real Life

LPC—The programming language used on LPMUDs, based on the C programming language

LPMUD—Lars Pengsl Multi-User Dungeon

LOL—Laughs Out Loud

MOO—MUD, Object Oriented

MUD—Multi-User Dungeon or Multi-User Dimension

MUF—Multi-User Forth. The programming language used on MUCKs

MUSH—Multi-User Shared Hallucination

OMW—On My Way (see *BRT*)

OOC—Out Of Character. Used to indicate conversation on topics outside of the MUD

PK—Player Killing

RL—Real Life

ROTFL or **ROFL**—Rolls on the Floor Laughing

RTFM—An offensive way of saying Read the Frigging Manual

SOC—South Of the Church

SOL—An offensive way of saying Surely Out of Luck

SP—Spell Points

WC—Weapon Class

WTF—An offensive way of saying What the Frig

WTG—Way To Go

XP—eXperience Points

Smileys

Smileys have been around on the Internet for a long time. They often are used on MUDs and by MUDders in other forms of communication (such as e-mail). The generic smiley, for example, often is used in a comment to designate sarcasm or humor. (On the Internet, sarcasm is not always as obvious as it may be in real life—sometimes its hard to guess what a person means when you cannot see his or her face or other body language.)

Following are some of the basics smileys. Many people have developed their own smileys and it also is possible to create new smileys by combining some of the ones in the following list:

`:-)` or `:)`	Generic smiley
`:-(` or `:(`	Sad
`;-)` or `;)`	Winking
`:-*`	Kissing
`:-P`	Sticking tongue out
`>:-)`	Devilish smiley
`:-D`	Very smiley
`:-o`	Look of surprise or "oh"
`8-)`	Wide eyed smiley
`=:-)`	Wild haired smiley
`:-9`	Licking its lips smiley
`B-)`	Smiley in sunglasses
`d:-)`	Smiley wearing a baseball cap

Other MUD Terms

Here is a quick list of other terms that may come up in the course of MUDding that you might want to know.

emotion—This often is used in e-mail and off-MUD conversation (talks, IRC, and so on) to portray a MUD emotion, such as *smile*

areas—On many MUDs, a specific wizard or group of wizards is responsible for maintaining and developing a region of the MUD. This region is called the wizard's area.

bug—A general computer term for a problem in a piece of software. There are bugs in MUDs just like in any software. Some bugs may be beneficial to players, others may be harmful. If you find a bug, you should report it to a wizard.

immortal—A wizard

lag—Noticeable delays when the machine or Internet connection the MUD is on begins to slow

link death—When your character is moved to a special room and held in storage with all of his or her items because your MUD connection (through telnet) dies through no fault of your own

maving—Accidently sending a private message to everyone (that is, mistyping a `tell` or `page` command so that it is sent to everyone in the room)

mortal—A player that can die (usually means not a wizard)

MUD sex—The act of having sexual relations on a MUD

MUDname—The pseudonym one uses on a MUD

newbie—A new MUD user

reboot—An impending reset of the MUD. This often means that you will lose all your equipment.

spamming—To send a large amount of data (usually via `say` or `tell`) to annoy other players or cause their connection to fail

spoofing—Sending fake `say` or `tell` messages so that they appear to be from someone else

TinySex—Term used on some MUSHes, MUCKs, and MOOs rather than the term "MUD sex"

wimpy—The setting in your character that controls when it will run from a fight

wiz—The act of becoming a wizard

C

APPENDIX

MUD Clients and Where to Find Them

The following list comes from the MUD FAQ, or list of frequently asked questions. This informational posting is maintained by Jennifer Smith (`jds@math.okstate.edu`) and is widely circulated on the Internet. The most current version can be retrieved via FTP from `ftp.math.okstate.edu:/pub/muds/misc/mud-faq`, and frequently is posted to MUD-related newsgroups. This list has the names and short descriptions of nearly all available client programs, as well as their standard FTP distribution sites.

UNIX Clients

Client:	TinyTalk
Comments:	Runs on BSD or SysV. Latest version is 1.1.7GEW. Designed primarily for TinyMUD-style MUDs. Features include line editing, command history, highlighting (whispers, pages, and users), gag, auto-login, simple macros, logging, and cyberportals.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
	`parcftp.xerox.com:/pub/MOO/clients`
	`ftp.tcp.com:/pub/mud/Clients`
Client:	TinyFugue
Comments:	Runs on BSD or SysV. Latest version is 3.2beta4. Commonly known as *TF*. Designed primarily for TinyMUD-style muds, although will run on LPMUDs and Dikus. Features include regexp highlights and gags, auto-login, macros, line editing, screen mode, triggers, cyberportals, logging, file and command uploading, shells, and multiple connects.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients/tf`
	`ftp.tcp.com:/pub/mud/Clients`
Client:	TclTT
Comments:	Runs on BSD. Latest version is 0.9. Designed primarily for TinyMUD-style MUDs. Features include regexp highlights, regexp gags, logging, auto-login, partial file uploading, triggers, and programmability.
Address:	`ftp.white.toronto.edu:/pub/muds/tcltt`
	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
Client:	VT
Comments:	Runs on BSD or SysV. Latest version is 2.15. Must have vt102 capabilities. Usable for all types of MUDs. Features include a C-like extension language (VTC) and a simple windowing system.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/vt`
	`ftp.tcp.com:/pub/mud/Clients`
Client:	LPTalk
Comments:	Runs on BSD or SysV. Latest version is 1.2.1. Designed primarily for LPMUDs. Features include highlighting, gags, auto-login, simple macros, logging.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/unixclients`
Client:	SayWat
Comments:	Runs on BSD. Latest version is 0.30beta. Designed primarily for TinyMUD-style MUDs. Features include regexp highlights, regexp gags, macros, triggers, logging, cyberportals, rudimentary xterm support, command line history, multiple connects, and file uploading.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`

Client:	PMF
Comments:	Runs on BSD. Latest version is 1.13.1. Usable for both LPMUDs and TinyMUD-style MUDs. Features include line editing, auto-login, macros, triggers, gags, logging, file uploads, an X-window interface, and capability to do Sparc sounds.
Address:	`ftp.lysator.liu.se:/pub/lpmud/clients` `ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
Client:	Tintin
Comments:	Runs on BSD. Latest version is 2.0. Designed primarily for Dikus. Features include macros, triggers, tick-counter features, and multiple connects.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
Client:	Tintin++
Comments:	Runs on BSD or SysV. Latest version is 1.5pl5. Derived and improved from Tintin. Additional features include variables, faster triggers, and a split-screen mode.
Address:	`ftp.princeton.edu:/pub/tintin++/dist` `ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
Client:	TUsh
Comments:	Runs on BSD or SysV. Latest version is 1.74. Features include highlighting, triggers, aliasing, history buffer, and screen mode.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
Client:	LPmudr
Comments:	Runs on BSD or SysV. Latest version is 2.7. Designed primarily for LPMUDs. Features include line editing, command history, auto-login, and logging.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`

EMACs Clients

Client:	MUD.el
Comments:	Runs on GNU Emacs. Usable for TinyMUD-style muds, LPMUDs, and MOOs. Features include auto-login, macros, logging, cyberportals, screen mode, and it is programmable.
Address:	`parcftp.xerox.com:/pub/MOO/clients` `ftp.math.okstate.edu:/pub/muds/clients/UnixClients`
Client:	TinyTalk.el
Comments:	Runs on GNU Emacs. Latest version is 0.5. Designed primarily for TinyMUD-style muds. Features include auto-login, macros, logging, screen mode, and it is programmable.
Address:	`ftp.tcp.com(128.95.10.106):/pub/mud/Clients` `ftp.math.okstate.edu:/pub/muds/clients/UnixClients`

Client: LPmud.el

Comments: Runs on GNU Emacs. Designed primarily for LPMUDs. Features include macros, triggers, file uploading, logging, screen mode, and it is programmable.

Address: `ftp.lysator.liu.se:/pub/lpmud/clients`

`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`

Client: CLPmud.el

Comments: Runs on GNU Emacs. Designed primarily for LPMUDs. Similar to LPmud.el, but with the added capability for remote file retrieval, editing in emacs, and saving, for LPMud wizards.

Address: `mizar.docs.uu.se:/pub/lpmud`

Client: MyMud.el

Comments: Runs on GNU Emacs. Latest version is 1.31. Designed primarily for LPMUDs and Dikus. Features include screen mode, auto-login, macros, triggers, auto-navigator, and it is programmable.

Address: `ftp.math.okstate.edu:/pub/muds/clients/UnixClients`

`ftp.tcp.com:/pub/mud/Clients`

VMS Clients

Client: tfVMS

Comments: VMS version of TinyFugue (see above). Uses Wollongong networking. Latest version is 1.0b2.

Address: `ftp.math.okstate.edu:/pub/muds/clients/VMSClients`

Client: TINT

Comments: Runs on VMS with MultiNet networking. Latest version is 2.2. Designed primarily for TinyMUD-style MUDs. Features include highlighting (whispers, pages, users), gags, file uploading, simple macros, and screen mode. See also *TINTw*.

Address: `ftp.math.okstate.edu:/pub/muds/clients/VMSClients`

Client: TINTw

Comments: Runs on VMS with Wollongong networking. See TINT.

Address: `ftp.math.okstate.edu:/pub/muds/clients/VMSClients`

`ftp.tcp.com:/pub/mud/Clients`

Client: DINK

Comments: Runs on VMS with either Wollongong or MultiNet networking. Similar to TINT. No longer supported by the author.

Address: `ftp.math.okstate.edu:/pub/muds/clients/VMSClients`

`ftp.tcp.com:/pub/mud/Clients`

Client:	FooTalk
Comments:	Runs on VMS with MultiNet networking and BSD UNIX. Primarily designed for TinyMUD-style MUDs. Features include screen mode, and it is programmable.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/VMSClients`
	`ftp.math.okstate.edu:/pub/muds/clients/UnixClients`

WinSock Clients (for Microsoft Windows)

Client:	WinMud
Comments:	Runs on MS Windows using WinSock. Primarily designed for LPs and DikuMUDs. Features include simple macros.
Address:	`ftp.cybernetics.net:/pub/users/lymang`
Client:	VWMud
Comments:	Runs on MS Windows using WinSock. Features include macros and triggers.
Address:	`ftp.primenet.com:/pub/users/kslewin`
Clients:	WinWorld
Comments:	Runs on MS Windows using WinSock.
Address:	`ftp.mgl.ca:/pub/winworld`
Client:	MUTT
Comments:	Runs on MS Windows using WinSock. Latest version is 01i. Name stands for Multi-User Trivial Terminal. Features include scripting, multiple connects, triggers, macros, logging, and so on.
Address:	`caisr2.cwru.edu:/pub/mud`
	`ftp.graphcomp.com:/pub/msw/mutt`
Client:	MudWin
Comments:	Runs on MS Windows using WinSock. Features include command history, simple macros, and logging.
Address:	`ftp.microserve.com:/pub/msdos/winsock`

Other Clients

Client:	REXXTALK
Comments:	Runs on IBM VM. Latest version is 2.1. Designed primarily for TinyMUD-style MUDs. Features include screen mode, logging, macros, triggers, highlights, gags, and auto-login. Allows some IBM VM programs, such as TELL and MAIL, to be run while connected to a foreign host.
Address:	`ftp.math.okstate.edu:/pub/muds/clients/misc`

Client:	MUDDweller
Comments:	Runs on any Macintosh. Latest version is 1.2. Connects to a MUD through either the communications toolbox or by MacTCP. Usable for both LPMUDs and TinyMUD-style MUDs. Current features include multiple connections, a command history, and a built-in MTP client for LPMUDs.
Address:	`rudolf.ethz.ch:/pub/mud`
	`mac.archive.umich.edu:/mac/util/comm`
	`ftp.tcp.com:/pub/mud/Clients`
Client:	Mudling
Comments:	Runs on any Macintosh. Latest version is 0.9b26. Features include multiple connections, triggers, macros, command line history, separate input and output windows, and a rudimentary mapping system.
Address:	`imv.aau.dk:/pub/Mudling`
	`ftp.math.okstate.edu:/pub/muds/clients/misc`
Client:	MUDCaller
Comments:	Runs under MS-DOS. Latest version is 2.50. Requires an Ethernet card, and uses the Crynwr Packet drivers. Does NOT work with a modem. (If you telnet in MS-DOS, you can probably use this.) Features include multiple connections, triggers, command-line history, scrollback, logging, macros, and separate input and output windows.
Address:	`ftp.tcp.com:/pub/mud/Clients`
	`ftp.math.okstate.edu:/pub/muds/clients/misc`
	`oak.oakland.edu:/pub/msdos/pktdrvr`

BSXMUD Clients

These clients run on various platforms, and enable the user to be able to see the graphics produced by BSXMUDs. BSXMUDs generally are LPMUDs (but not necessarily) that have been hacked to allow the sending of polygon graphics coordinates to BSXclients, thus enabling you to play a graphic MUD rather than just a text-based MUD.

Comments:	For Amiga: Modem or TCP/IP (`AmigaBSXClient2_2.lha`)
	For PC: Requires a modem (`msclient.lzh AND x00v124.zip`)
	For X11: Sources, version 3.2 (`bsxclient3_8c.tar.Z`)
	For Sun4: Binary (`client.sparc.tar.Z`)
	Also available are programs to custom draw your own graphics for a BSXMUD (`muddraw.tar.gz, bsxdraw.zoo`)
Address:	`ftp.lysator.liu.se:pub/lpmud/bsx`
	`ftp.math.okstate.edu:/pub/muds/BSXstuff`

D APPENDIX

Available Servers

There are many MUD servers available. While it is beyond the scope of this book to discuss them all, it might be of interest to you to find out what the different servers are. The following list, compiled by Jennifer Smith, is from the MUD FAQ and shows a comprehensive list of known MUD servers, a brief description of each, and an FTP site where each can be found.

MUD FAQ Note

Just because we say something's available doesn't mean we have it. Please don't ask us; ask around for FTP sites that might have it, or try looking on `ftp.tcp.com` or `ftp.math.okstate.edu`.

Combat-Oriented MUDs

Server:	MUD
Comments:	The original, by Richard Bartle and Roy Trubshaw, written back in 1978. An advanced version of MUD2 is now running on CompuServe under the name of "British Legends." A few MUD2s can still be found running here and there. The three known ones are `portal.aladdin.co.uk`, `craic.iol.ie`, and Iplay Online at `199.182.210.2`. Source generally not available.
Server:	AberMUD
Comments:	One of the first adventure-based MUDs. Players cannot build. In later versions, a class system was added, and wizards can build onto the database. It's named after the university at which it was written, Aberystwyth. Latest version is 5.21.5. Supports all the usual in combat game design, including BSX graphics and MUDWHO. Not too big, and it will run under BSD and SYSV. Amiga TCP/IP support now included.
Address:	`A.Cox@swan.ac.uk.` `sunacm.swan.ac.uk:/pub/misc/AberMUD5/SOURCE`
Server:	LPMUD
Comments:	The most popular combat-oriented MUD. Players cannot build. Be warned, though: LPMUD servers version 3.* themselves are very generic—all of the universe rules and so forth are written in a separate module, called the MUDlib. Most LPMUDs running are written to be some sort of combat system, which is why I've classified them here, but they don't have to be! Wizards can build onto the database, by means of an object-oriented C-like internal language called LP-C. It's named after its primary author, Lars Pensj¦. Latest version is 3.2, AKA Amylaar. Fairly stable, and size varies from medium to large. Driver (server) versions seem to have split into several main variants, not counting possible MUDlibs (databases) available. Amylaar, CD, and MUDOS are the current favorites. For further information, e-mail to `amylaar@meolyon.hanse.de`. There is a port of 3.1.2 for Amigas, called aMUD, now included in LPMUD v3.2. For further information, e-mail to `mateese@ibr.cs.tu-bs.de`. See the `rec.games.MUD.lp` FAQ for more info.

Address:	`ftp.lysator.liu.se:/pub/lpMUD`
	`ftp.cd.chalmers.se:/pub/lpMUD/cdlib`
	`ftp.tu-bs.de:/pub/games/lpMUD`
	`ftp.ccs.neu.edu:/pub/MUD/drivers/MUDos`
	There is a port of 3.1.2 for MSDOS, that requires at least a 386 to run. It accepts connections from serial ports.
Address:	`ftp.ccs.neu.edu:/pub/MUD/drivers/lpMUD/msdos`
Server:	DGD
Comments:	Written by Felix Croes. A reimplementation from scratch of the LPMUD server. It is disk-based, and thus uses less memory. It's also smaller and lacks many of the features of the other LPMUD servers, though it is capable of simulating most of those features in LPC. Many DGDs are simulating an LP, but there are several MUDs that now use DGD to simulate a MOO variant. The name stands for Dworkin's Generic Driver. Stable. Has been ported to Atari ST and Commodore Amiga.
Address:	`ftp.lysator.liu.se:/pub/lpMUD/drivers/dgd`
Server:	DikuMUD
Comments:	Newer than LPMUD, and gaining in popularity. Almost identical from the players' point of view. Uses a guild system instead of a straight class system. Wizards can add on to the database, but there is no programming language, as in LP. It's named after the university at which it was written, Datalogisk Institut Koebenhavns Universitet (Dept. of Datalogy, University of Copenhagen).
Address:	`coyote.cs.wmich.edu:/pub/Games/DikuMUD`
Server:	YAMA
Comments:	PC MUD writing system, using `waterloo wattcp`. Runs on a 640Kb PC/XT or better. Runs best with about a 1Mb RAM disk, but is fine without. A separate Windows version (yamaw) runs under Windows and allows you to run a MUD on a 286 or higher without taking over the machine.
Address:	`sunacm.swan.ac.uk:/pub/misc/YAMA`
Server:	UriMUD
Comments:	Developed from an LPMUD2.4.5, the code structure is very similar. Features include better speed, flexibility, stronger LPC, and the ability to handle multiple MUDlibs under one parser. Latest version is 2.5.
Address:	`uriMUD.isp.net:/uriMUD/src`
Server:	Ogham
Comments:	From the players' point of view, similar to LPMUD. No programming language or database, as MUD compiles to a single binary executable. Latest version is 1.5.
Address:	`ftp.ccs.neu.edu:/pub/MUD/servers/ogham`
	`ftp.math.okstate.edu:/pub/MUDs/servers`

Server: CircleMUD

Comments: Derivative of DikuMUD Gamma v0.0. Developed by Jeremy Elson (`jelson@cs.jhu.edu`). Less buggy and tighter code all in all. Latest version is 2.20. Also see URL `http://www.cs.jhu.edu/other/jelson/circle.html`

Address: `ftp.cs.jhu.edu:/pub/CircleMUD`

`sunsite.unc.edu:/pub/Linux/games/MUDs`

`ftp.math.okstate.edu:/pub/MUDs/servers`

Server: AmigaMUD

Comments: Written from scratch for Commodore Amiga computers. Includes custom client which supports graphics and sound. Disk-based, fast programming language, standard scenario including built-in mail and bboards. Obtained from the Aminet FTP sites.

Address: `ftp.wustl.edu:/pub/aminet/game/role/AMClnt.lha, AMSrv.lha`

TinyMUD-Style MUDs

Server: TinyMUD

Comments: The first, and archetypical, socially-oriented MUD. It was inspired by and looks like the old VMS game Monster, by Rich Skrenta. Players can explore and build, with the basic `@dig`, `@create`, `@open`, `@link`, `@unlink`, and `@lock` commands. Players cannot teleport, and couldn't use `@chown` or set things `DARK` until later versions. Recycling didn't exist till the later versions, either. It's called "Tiny" because it is—compared to the combat-oriented MUDs. Original code written by Jim Aspnes. Last known version is 1.5.5. Not terribly big, and quite stable.

Address: `ftp.math.okstate.edu:/pub/MUDs/servers`

`primerd.prime.com:/pub/games/MUD/tinyMUD`

There is a PC port of TinyMUD, along with some extra code. It accepts connections from serial ports.

Address: `ftp.tcp.com:/pub/MUD/TinyMUD`

There is a modified version of TinyMUD called PRISM, that works for PCs, Atari STs, and most UNIXes. It also comes with an internal BSX client for MS-DOS.

Address: `lister.cc.ic.ac.uk:/pub/prism`

Server: TinyMUCK v1.*

Comments: The first derivative from TinyMUD. Identical to TinyMUD, except that it added the concept of moveable exits, called `@actions`. Also introduced the `JUMP_OK` flag, which allows players to use `@teleport`, and `@recycle`, which TinyMUD later added. Its name, MUCK, is derived from MUD, and means nothing in particular. Original code written by Stephen White. Latest stable version is 1.2.c&r, which brought TinyMUCKv1 up to date with later TinyMUD things. Not terribly big.

Address: `ftp.math.okstate.edu:/pub/MUDs/servers`

Server:	TinyMUSH
Comments:	The second derivative from TinyMUD. Also identical to TinyMUD, with the addition of a very primitive script-like language. Introduced `JUMP_OK` like TinyMUCK, and has recycling, except it is called `@destroy`. Also introduced the concept of puppets, and other objects that can listen. In later versions the script language was extended greatly, adding math functions and many database functions. In the latest version, 2.0.*, it's gone to a disk-basing system as well. Its name, MUSH, stands for Multi-User Shared Hallucination. Original code written by Larry Foard. The latest non-disk-based version is PennMUSH1.50p10g5, which is quite similar to 2.0 from the user's point of view. Both the disk-based version and the non-disk-based version are being developed at the same time (PennMUSH is now being developed under the name PennMUSH-Dune). TinyMUSH is more efficient in some ways than TinyMUD, but winds up being larger because of programmed objects. Version 2.0 generally uses less memory but a great deal more disk space. 2.0 may also be able to be run under VMS, as well as both BSD and SysV UNIX. Most recent version is 2.0.10p6.

Address:	`caisr2.caisr.cwru.edu:/pub/mush`
	`mellers1.psych.berkeley.edu:/pub/DuneMUSH/Source`
	`ftp.cis.upenn.edu:/pub/lwl`
	`primerd.prime.com:/pub/games/MUD/tinymush`
	`ftp.tcp.com:/pub/MUD/TinyMUSH`

Server:	TinyMUCK v2.*
Comments:	TinyMUCKv1.* with a programming language added. The language, MUF (multiple user FORTH), is only accessible to people with the `MUCKER` flag. Changed the rules of the `JUMP_OK` flag somewhat, to where it's nice and confusing now. MUF is very powerful, and can do just about anything a wizard can. Original version 2.* code written by Lachesis. Latest version is 2.3b, with several varieties (FBMUCK and DaemonMUCK 0.14 the most common). The name doesn't mean anything. Can be quite large, especially with many programs. Mostly stable.
Address:	`ftp.tcp.com:/pub/MUD/TinyMUCK`
Server:	TinyMUSE
Comments:	A derivative of TinyMUSH. Many more script-language extensions and flags. Reintroduced a class system, a la combat-oriented MUDs. The name stands for Multi-User Simulation Environment. Latest version is 1.7b4. Not very stable.
Address:	`mcmuse.mc.maricopa.edu:/muse/server` `caisr2.caisr.cwru.edu:/pub/mush/muse`

Server:	TinyMAGE
Comments:	The bastard son of TinyMUSH and TinyMUCK. It combines some of MUSH's concepts (such as puppets, `@adesc/@asucc`, several programming functions, and a few flags) with TinyMUCK2.*x*. Interesting idea; really busted code. The name doesn't mean anything. Latest version is 1.1.2.
Address:	`ftp.tcp.com:/pub/MUD/TinyMAGE`
Server:	MUG
Comments:	Derivative of TinyMUD 1.4.1. It's name stands for Multi-User Game. Powerful but awkward programming language, which is an extension of the user language; primitive notion of puppets; inheritance; sane variable/property matching; arrays and dictionaries in hardcode. Somewhat non-standard and buggy in a few places. Requires gcc.2.4.5 or greater (or other good C++ compiler) to compile.
Address:	Available by e-mail from `wizard@cs.man.ac.uk`; Development site is UglyMUG (`wyrm.cs.man.ac.uk 6239`)
Server:	TeenyMUD
Comments:	A TinyMUD clone, written from scratch. Its main feature is that it is disk-based. Original code written by Andrew Molitor. Latest version is 1.3. Very small, and mostly stable.
Address:	`fido.econ.arizona.edu:/pub/teeny`

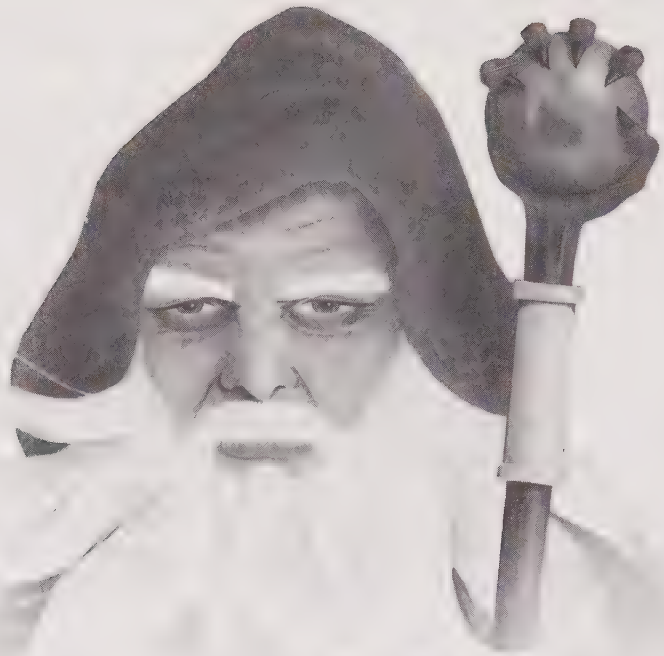
Miscellaneous

Server:	UberMUD
Comments:	The first MUD where the universe rules are written totally in the internal programming language, U. The language is very C/Pascal-like. The permissions system is tricky, and writing up every universe rule (commands and all) without having big security holes is a pain. But it's one of the most flexible MUDs in existence. Great for writing up neat toys. It's also disk-based. Original code written by Marcus J. Ranum. Latest version is 1.13. Small in memory, but can eat up disk space. Quite stable.
Address:	`decuac.dec.com:/pub/MUD` `ftp.white.toronto.edu:/pub/MUDs/uber` `ftp.math.okstate.edu:/pub/MUDs/servers`
Server:	MOO
Comments:	An Object-Oriented MUD. Unfortunately, the first few versions weren't fully object-oriented. Later versions fixed that problem. There is a C-like internal programming language, and it can be a bit tricky. Original code written by Stephen White. Last version is 2.0a.
Address:	No Known Site

Server:	LambdaMOO
Comments:	An offshoot of MOO. Added more functionality, many new features, and a great deal more stability, in a general rewrite of the code. This is the only version of MOO that is still being developed, by Pavel Curtis. Latest version is 1.7.8p3.
Address:	`parcftp.xerox.com:/pub/MOO`
Server:	SMUG
Comments:	Also known as TinyMUD v2.0. It has an internal programming language, and it does have some inheritance. Surprisingly similar to MOO in some ways. SMUG stands for Small Multi-User Game. Original code written by Jim Aspnes.
Address:	`ftp.tcp.com:/pub/MUD/Smug`
Server:	UnterMUD
Comments:	A network-oriented MUD. It's disk-based, with a variety of db layers to choose from. An UnterMUD can connect directly to other UnterMUDs, and players can carry stuff with them when they tour the Unterverse. This can be a bit baffling to a new user, admittedly, but those people already familiar with the old cyberportals and how they work (invented way back with the original TinyMUD) will adjust to the new, real, cyberportals easily. There is both a primitive scripting language and much of the U language from UberMUD built in, as well as a combat system that can be compiled in if wanted. The parsing can be a bit odd, especially if you're used to the TinyMUD-style parser. Unter is also the only MUD that can run under BSD UNIX, SysVr4 UNIX, and VMS with MultiNet networking, with little to no hacking. Original code written by Marcus J. Ranum. Latest version is 2.1. Small in memory, but can eat up a lot of disk space.

Address:	`ftp.math.okstate.edu:/pub/MUDs/servers`
	`decuac.dec.com:/pub/MUD`
	`ftp.tcp.com:pub/MUD/UnterMUD`

I
INDEX



Book ISBN 0-672-30723-5

Basic MUSH Commands

Command	*Abbreviation*	*Description*
`act` *`message`*	`:`*`message`*	Acts the given *`message`*
`drop` *`object`*		Drops the *`object`*
`examine` *`object`*		Obtains detailed *`object`* info
`get` *`object`*		Takes the *`object`*
`go` *`direction`*	*`direction`*	Moves in the given *`direction`*
`help`		Accesses help facilities
`home`		Returns to your home room
`inventory`	`i`	Inventory
`look`		Looks at room description
`look` *`object`*		Looks at *`object`* description
`page` *`player`* `=` *`message`*	`p` *`player`*`=`*`message`*	Pages *`player`* with *`message`*
`QUIT`		Quits playing the MUD
`say` *`message`*	`"`*`message`*	Speaks the given *`message`*
`take` *`object`*		Takes the *`object`*
`whisper` *`player`* `=` *`message`*	`w` *`player`*`=`*`message`*	Whispers *`message`* to *`player`*
`WHO`		Sees who else is playing

Explore the Depths of the Internet's Multi-User Dungeons!

Your quest has begun! It's your turn to wander the corridors of the Net and unlock the magical, mystical powers of MUDs, MOOs, MUCKs, and MUSHes.

Let your imagination run wild with *Secrets of the MUD Wizards!*

- **Unlock the secrets on how to become a wizard; walk through the process of programing objects and rooms, and administering a MUD**
- **Discover the best MUD sites in the universe with the MUD directory**
- **Explore all the commands and how to use them to your best advantage**
- **Uncover the differences between social MUDs and combat MUDs**
- **Take a journey through the various MUDs, MOOs, and MUCKs—in detail**
- **Wander through real-life stories of successes and failures, relationships, and MUD marriages**
- **Create your own fantastic world and program and play new MUDs**

Andrew Busey has over seven years of Internet experience, several of which were spe MUDding. For hree years, Tarod (B MUD name) h n one of the go popular LPM own as Realms. Ta s also held Wizard ns on several othe s and played on e re.

GUARANTEED
Jennifer Smith
TIMELY & ACCURATE

$25.00 USA
$34.95 CAN
£19.50 Net UK

ISBN 0-672-30723-5
90000
9 780672 307232

User Level

New • Casual • Accomplished • Expert

sams.net

Category: Communications/Online—Internet/Entertainment